MMXSNMP:

A Library for SNMP Control of the MultiMedia eXplorer

Jeremy Elson

jeremy.elson@nih.gov

Computational Bioscience and Engineering Laboratory
Division of Computer Research and Technology
National Institutes of Health
Bethesda, MD 20892 USA

17 October 1997

Table of Contents

List of Figures

List of Example Programs

1 Introduction

The **M**ulti**M**edia e**X**plorer (MMX) is an Asynchronous Transfer Mode (ATM) desktop multimedia interface that provides users with advanced simultaneous bi-directional video and audio communications capabilities [1]. The stand-alone MMX, sold by STS Technologies, Inc. (Bridgeton, MO), requires no resources of the host PC or workstation for processing of the multimedia streams. Instead, audio and video processing are performed by the MMX, allowing the host system to operate at full performance.

The MMX can be controlled via SNMP [2], the Simple Network Management Protocol. SNMP commands can be used to start and stop the transmission and reception of multimedia streams, set the video compression factor and audio volume, and make other changes to the MMX's configuration. The NIH MMXSNMP Library provides a high-level C++ API to the MMX's features, hiding the low-level details of the SNMP commands from applications programmers. This speeds the development of applications that wish to control the MMX.

General information about the MMX can be found in the MMX Installation and Operation Manual [3]. This document is meant to be both a supplement to that manual and a complete description of the MMXSNMP API.

1.1 Notice and Disclaimer

WARE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT SHALL THE FEDERAL GOVERNMENT BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE SOFTWARE TO OPERATE WITH ANY OTHER SOFTWARE).

IF ANY MODIFICATIONS ARE MADE TO THIS SOFTWARE BY THE RECIPIENT OR A THIRD PARTY, THE SOFTWARE MAY NO LONGER BE REFERRED TO AS THE "NIH MMXSNMP LIBRARY" AND SUCH MODIFICATIONS ARE STRONGLY DISCOURAGED. SUGGESTIONS FOR ENHANCEMENTS, CORRECTIONS, OR MODIFICATIONS TO THE SOFTWARE OR ITS DOCUMENTATION MAY BE SUBMITTED TO THE GOVERNMENT TO BE CONSIDERED FOR INCLUSION INTO FUTURE REVISIONS OF THIS SOFTWARE. THE GOVERNMENT IS, HOWEVER, UNDER NO OBLIGATION TO PRODUCE FUTURE REVISIONS, INCORPORATE CHANGES, OR MAKE CORRECTIONS.

1.2 Features of the Library

MMXSNMP is a simple, well-documented library of C++ functions that can be used to control the configuration of the MMX. Although it is possible to control the MMX directly by using a generic SNMP library instead of MMXSNMP, MMXSNMP adds features that make writing applications much easier. Some operations require a number of complex SNMP messages to be sent and received; this complexity has been completely hidden from the user of the library.

For example, consider what is required if we want the MMX to begin transmitting video on a particular VPI and VCI. Doing this with a "plain vanilla" SNMP library requires the following steps:

- Get a list of the current video streams that exist in the MMX's video connection table; this is accomplished by sending a series of SNMP GETNEXT requests to the MMX to retrieve the SNMP object identifiers of each connection that exists in the table.
- For each object identifier reported in the previous step, check the value of the object itself to determine if the connection it represents is active.
- If one of the active streams found in the previous step is a video transmit stream, close the stream.
- Check if an idle video stream with the desired VPI and VCI of the new stream is already in the connection table. If not, create an entry in the table by changing its status from "closed" to "idle".
- Set the Q Factor (video compression factor) of the idle stream with the desired VPI and VCI.
- Select the analog video input source for the idle stream with the desired VPI and VCI.
- Make the idle stream active.

MMXSNMP makes this complex task much easier by providing a simple function (`video_start()`) that automatically performs all of the above steps. Most of the common configuration functions—such as starting or stopping video, starting or stopping audio, changing the video Q Factor or video input source, or setting the master audio volume—can also be accomplished by calling a single function. The MMXSNMP API was designed to be simple, consistent, and easy to learn and to use.

MMXSNMP is written in C++. An application that wishes to control an MMX simply declares an instance of a C++ class named `mmxsnmp`, passing the Internet hostname of the MMX to the constructor (after ensuring that IP connectivity exists between the workstation and MMX to be controlled, as described in Section 3). The MMX can then be controlled by calling member functions of the instance. For example, this simple code might be used to set the audio sampling rate and video input port of the MMX `mmx1.dcrt.nih.gov`:

```
#include "mmxsnmp.h"

main()
{
   mmxsnmp my_mmx("mmx1.dcrt.nih.gov");

   my_mmx.audio_sample_rate(samp44K);
   my_mmx.video_input(SIN1);
}
```

More than one MMX can be simultaneously controlled simply by declaring more than once instance of the `mmxsnmp` class. For example, assuming that the appropriate PVCs already exist in the ATM network, two MMXs could be instructed to start and stop sending video to each other using the following code:

```
#include "mmxsnmp.h"

main()
{
   mmxsnmp mmx1("mmx1.dcrt.nih.gov");
   mmxsnmp mmx2("mmx2.dcrt.nih.gov");

   // Set up video from mmx1 to mmx2 using a Q Factor of 30
   mmx1.video_q_factor(30, TRANSMIT);
   mmx1.video_start(vc(0, 100, TRANSMIT)); // Transmit on VPI 0, VCI 100
   mmx2.video_q_factor(30, RECEIVE);
   mmx2.video_start(vc(0, 101, RECEIVE));  // Receive on VPI 0, VCI 101

   // Set up video from mmx2 to mmx1 using a Q Factor of 75
   mmx2.video_q_factor(75, TRANSMIT);
   mmx2.video_start(vc(0, 200, TRANSMIT)); // Transmit on VPI 0, VCI 200
   mmx1.video_q_factor(75, RECEIVE);
   mmx1.video_start(vc(0, 201, RECEIVE));  // Receive on VPI 0, VCI 201

   // ...some time later, we want to shut down the video transmission
   mmx1.video_stop(TRANSMIT); // Stop transmitting video
   mmx1.video_stop(RECEIVE);  // Stop receiving video
   mmx2.video_stop(BIDIR);    // BIDIR is a shortcut to turn off both TRANSMIT and
                              // RECEIVE at the same time
}
```

Setting up audio connections and changing other configuration parameters is just as simple. Parameters such as the Q Factor and video input source can be changed either before a video stream is activated, or while it is already active. The exact semantics of `mmxsnmp`'s member functions are all documented in the Reference Manual (Appendix B).

1.3 Known Limitations

As mentioned in the previous section, a number of MMXSNMP's functions are not atomic; they require a series of SNMP commands to be sent and received. Because the MMX does not provide any way of granting exclusive write access to a single entity, MMXSNMP must simply assume that it has exclusive access to the MMX's configuration when attempting to modify it. If more than one application is simultaneously sending and receiving SNMP commands to the MMX, the result is a race condition that may cause the MMX to be left in an inconsistent state. Therefore, it is the responsibility of the application to ensure that there is never more than a single instance of MMXSNMP simultaneously attempting to make modifications to the configuration of an MMX.

There are a number of "mandatory SNMP objects," or objects that must be supported by all SNMP-manageable devices. Although the MMX supports all of these objects, MMXSNMP does not have an interface for accessing most of them. This is because it was deemed more important to simplify the MMXSNMP API as much as possible, rather than cluttering it with esoteric commands. Unsupported SNMP objects include most of the System Group (`sysUpTime`, `sysContact`, `sysName`, etc.) and the entire Interfaces Group (`ifNumber`, `ifIndex`, `ifAdminStatus`, etc.). Of course, these objects can still be accessed with some other SNMP tool or library if necessary. The complete specification of the MMX's SNMP management capabilities can be found in the MMX MIB [4].

There are certain MMX configurations that are legal according to the definitions in the MMX MIB, but which are not supported by MMXSNMP. All of these are cases in which the feature was too esoteric to justify making the interface more complex. They are:

- If the MMX is configured to receive multiple audio streams (operating in `MODE1`), the MMX MIB allows each incoming audio stream to have its own setting for the master volume and the feedback volume. (The feedback volume is the percentage of the local audio input which is played on the local audio output.) MMXSNMP treats the master volume and feedback volume as global values.
- The MMX MIB allows the user to independently set the feedback volume for the left and right channels. MMXSNMP uses the same feedback volume for both channels.
- The MMX MIB requires that the audio mix table and feedback volume be specified on a scale from 0...255; MMXSNMP requires that these values be specified on a scale from 0...100. This provides slightly less granularity in the selection of the audio mix table and feedback volume.[1]

Parts of MMXSNMP are implemented using C++ derived exceptions. Unfortunately, this feature is not yet well-supported by many C++ implementations. MMXSNMP was developed using gcc 2.7.2.3, which includes limited but sufficient support for derived exceptions; it is described as "alpha–test code" in the gcc documentation. Users of gcc 2.7.x should be aware that all code that uses MMXSNMP library calls, and MMXSNMP itself, must be compiled using the `-fhandle-exceptions` and `-frtti` compiler flags. Versions of gcc before 2.7.x may not work at all. gcc 2.8.0, when released, will have better exception support. In addition to the above limitations, MMXSNMP supports only very basic error reporting in the form of exceptions (see Section 4.5).

MMXSNMP was developed on a Sun SparcStation 20 under Solaris 2.5.1 using gcc 2.7.2.3. Although MMXSNMP was written without the use of any Solaris-specific operating system features or gcc-specific language extensions, it is not currently known how well the library will work under other combinations

[1] The goal was for the API to use the same scale for all audio volume specifiers; the MMX MIB requires that the audio master volume be specified on a 0...100 scale. The decision was made to reduce the granularity of the audio mix table and feedback volume instead of artificially inflating the granularity of the master volume scale because the 0...100 scale is more intuitive than the 0...255 scale.

of hardware, operating systems, and compilers. If compatibility problems are found, future versions of the library may be released that use the GNU `autoconf` utility to automatically detect the environment in which the library is being used.

1.4 Downloading and Installing the Library

MMXSNMP is available on the Internet at `http://cbel.dcrt.nih.gov/~jelson/mmxsnmp`. The distribution includes the library itself, example code using the library, and this documentation.

Note that while the MMXSNMP's source code was written at the National Institutes of Health, the MMXSNMP distribution includes a copy of the source code for the SNMP library distributed by the Electrical Engineering Department of the University of California at Davis. This is because MMXSNMP uses the UCD SNMP library to perform the actual low-level SNMP communication. The UCD SNMP library is included with the MMXSNMP distribution as a convenience, so that users are not required to download it separately.

This section will contain more detailed instructions on compiling and installing MMXSNMP when those details are available.

1.5 Acknowledgments

Most of the information about the MMX's operation came from various documents provided by STS Technologies, Inc. (Bridgeton, MO), including the MMX Installation and Operation Manual [3] and the MMX MIB [4]. Most of Section 5 was taken directly from STS Technologies' Draft Working Note [5] on that topic. Additional clarifications were provided by Scott Chaney of STS Technologies and John DeHart of the Department of Computer Science at the Washington University in St. Louis.

The NIH MMXSNMP Library was made possible by a large number of high-quality, freely available software packages. Many of the tools used in the development of this software, including the compiler itself, were written by the Free Software Foundation. This document was produced using LaTeX (which, in turn, uses TeX).

MMXSNMP uses the free SNMP library distributed by the Electrical Engineering Department of the University of California at Davis (version 3.2). Their library, in turn, was originally based on the Carnegie Mellon University SNMP implementation (version 2.1.2.1). More information about the UCD SNMP library can be found at `ftp://ftp.ece.ucdavis.edu/pub/snmp`.

The Reference Manual (Appendix B) was produced with the help of DOC++, a free C++ documentation system in the style of JavaDoc, written by Roland Wunderling and Malte Zöckler. More information about DOC++ can be found at `http://www.zib.de/Visual/software/doc++/index.html`.

This library is also freely available, in the spirit of the other free software that made it possible.

2 Using the MMX in an ATM Network

2.1 ATM Network Topology

A typical MMX configuration within an ATM network is shown in Figure 1. Each MMX M_i is attached to a port P_i of an ATM switch S_i. We will use the notation (S_i, P_i) to refer to the port and switch to which MMX M_i is attached. Figure 1 shows a general case with each MMX on its own switch; of course, more than one MMX can be connected to the same switch. It is also not required that each

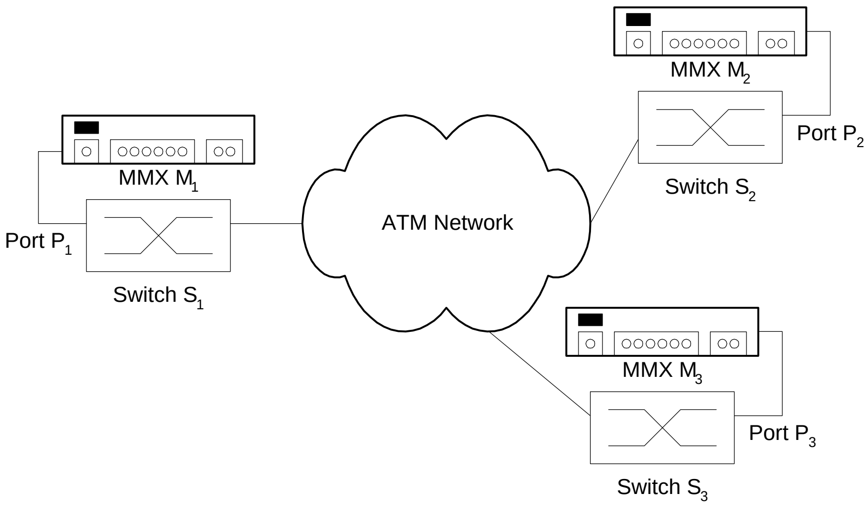

Figure 1: MMXs in an ATM Network

MMX be attached to a unique port (i.e., not every pair (S_i, P_i) must be unique). The switches can be directly attached to each other, or attached through any intermediate public or private ATM network of arbitrary complexity, as long as some method for creating calls through that network exists.

2.2 Establishing Communication Streams

The MMX can almost be considered a "dumb terminal" in relation to the ATM network; it does not support most of the protocols and standards typically used by ATM end-stations. For example, it does not support ILMI and does not exist within any ATM address space. Because it does not have an ATM address, the MMX naturally also cannot support any type of UNI signaling [6]; this means that it cannot request calls from an ATM network.

The MMX is essentially capable of only two functions. The first is digitizing its local analog audio and/or video inputs and encoding them in a cell stream, which it then transmits on its ATM interface using a user-selected VPI and VCI. The second is the inverse operation: watching its incoming cell stream for cells marked with a user-selected VPI and VCI, and reconstructing those cells into a local analog audio or video output. It should again be emphasized that the limit of the MMX's knowledge when transmitting or receiving a multimedia stream is the *local* VPI and VCI on which the stream is being transmitted or received;[2] the MMX has no knowledge of what happens to the cells after they leave its interface. Transmitting a video stream *to a particular ATM address* is a fundamentally different level of functionality, not supported by the MMX.

Given the above limitations, there are at least three steps involved in setting up communication between two MMXs. For example, if a user wishes to transmit a one-way video stream from MMX M_1 to MMX M_2, the user (or an application) must:

1. Tell MMX M_1 to begin transmitting video on some VPI:VCI (i, j).
2. Set up a point-to-point call in the ATM network so that all cells originating at (S_1, P_1) with VPI:VCI (i, j) are routed to (S_2, P_2) with a (possibly different) VPI:VCI (i', j').
3. Tell MMX M_2 to begin receiving video on VPI:VCI (i', j').

Steps 1 and 3 can be accomplished using MMXSNMP, which in turn uses SNMP commands to request that the MMX start transmitting or receiving the video streams. Step 2, creating the call in the ATM network, must be accomplished through some other means—for example, using an ATM network management application, or using a workstation's UNI signaling capabilities as described in Section 2.3. Note that if M_1 and M_2 are using the same switch port, they must be sending and receiving using a distinct VPI and VCI; in other words, $(S_1, P_1) = (S_2, P_2) \Longrightarrow (i, j) \neq (i', j')$.

It should be noted that the above steps only create a one-way video stream from a single source to a single destination. For a single video stream to be received by more than one destination, Step 2 should be modified to create a point-to-multipoint ATM call instead of a point-to-point call; this way, video cells generated by the source MMX are replicated by the ATM network itself, and copies are sent to each desired destination MMX.

To create a full-duplex video stream, allowing two conference participants to simultaneously see each other, a more complex procedure is required:

1. Tell MMX M_1 to begin transmitting video on some VPI:VCI (i_1, j_1); tell MMX M_2 to begin transmitting video on some VPI:VCI (i_2, j_2).

[2] Recall that in an ATM network, VPI and VCI identifiers have only local significance; the cells belonging to a call are usually transmitted on a different VPI and VCI through each link in the network. [7]

2. Set up two calls in the ATM network such that all cells originating at (S_1, P_1) with VPI:VCI (i_1, j_1) are routed to (S_2, P_2) with a VPI:VCI (i'_1, j'_1); all cells originating at (S_2, P_2) with VPI:VCI (i_2, j_2) are routed to (S_1, P_1) with a VPI:VCI (i'_2, j'_2).
3. Tell MMX M_1 to begin receiving video on VPI:VCI (i'_2, j'_2); tell MMX M_2 to begin receiving video on VPI:VCI (i'_1, j'_1).

Notice that M_1 has no knowledge of M_2 or vice-versa; each MMX does not even know on which VPI:VCI the other is transmitting or receiving its video stream. It is the responsibility of application software to coordinate MMXs both with each other and with the ATM network, as well as providing a user interface. MMXSNMP can be used as part of such application software to control the MMX itself; however, other services, such as ATM call setup and some form of interprocess communication, will also be required by that application. These additional services are *not* provided by MMXSNMP or the MMX itself.

2.3 Workstations in the ATM Network

Sites using MMXs often also use workstations with ATM interfaces. The MMX has a *host port* which allows an MMX and a workstation to share a port on the ATM switch, as shown in Figure 2. When the MMX's host port is enabled, it forwards all cells that it receives from the switch to the workstation, in addition to monitoring the cell stream for any cells marked with one of the VPI:VCI pairs that it has been configured to receive as an audio or video stream. The MMX also combines any cells that it may be generating for transmitting audio and video with cells that the workstation is transmitting, forwarding the blended cell stream to the switch as if the entire stream had been generated by a single end-station. Special care must be taken to configure the workstation and MMX so that the combined traffic does not exceed the capacity of the link from the switch to the MMX.

Interposing the MMX in between the workstation and switch has an important additional advantage (other than the obvious advantage of using fewer switch ports). We mentioned in Section 2.2 that the MMX does not have its own ATM address; this forces us to reference the MMX by both the address and physical port number of the switch to which the MMX is attached. This is undesirable because application software which automatically creates calls for the MMX will need to be reconfigured every time any MMX is moved to a different port or switch. On the other hand, ATM workstations are typically full-featured ATM end-stations that have real ATM addresses and are capable of registering those addresses with the network using ILMI. If the workstation and MMX share a switch port, it is possible to refer to the MMX using workstation's ATM address.

We also mentioned in Section 2.2 that the MMX is not capable of setting up its own calls through the network, but a workstation behind the MMX could use its UNI services to request calls on the MMX's behalf. This would avoid the use of a separate network management application to create MMX calls.

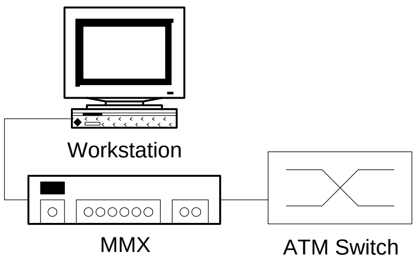

Figure 2: Workstation using the MMX's Host Port

3 Establishing SNMP Connectivity

3.1 Introduction

Before MMXSNMP can be used, the MMXs and host workstations must be configured so that SNMP commands can flow between them. In addition to the audio and video cells that it transmits, the MMX is also designed to simultaneously support a single point-to-point IP-over-ATM link between itself and another ATM end-station. Once this link has been established, higher-layer protocols and services become available, including SNMP (which uses UDP/IP), and responding to `ping` requests (ICMP `ECHO_REQUEST` datagrams).

The MMX's IP services must be configured using its serial console port. Details of how to use the console can be found in the MMX's Installation and Operation Manual [3]. Once IP connectivity has been established, most other configuration can be performed either by using `telnet` to connect to the MMX, or by using MMXSNMP.

The MMX supports IP-over-ATM using a small subset of the standard defined in RFC1577, "Classical IP and ARP over ATM" [8]. Due to the fact that the MMX does not have a true ATM address and cannot perform UNI signaling, the MMX is not capable of joining a typical "plug-and-play" network of Classical IP workstations. Instead, a single point-to-point link between an MMX and a workstation must be manually configured, as described in Section 3.2. *Only* that workstation will then be capable of communicating directly with the MMX; however, as discussed in Section 3.3, a workstation may be directly connected to more than one MMX at the same time.

If it is absolutely required that a single MMX be controlled by more than one workstation, it is possible to give workstations an indirect IP connection to an MMX if the directly connected workstation is configured as an IP router. However, such a configuration is generally not recommended because it is complex and difficult to manage. This topic is covered in Section 3.4.

3.2 Configuring Classical IP

In general, there are 4 steps involved in establishing the IP-over-ATM link between an MMX and a workstation. The administrator must:

1. Choose a unique IP address for the MMX, preferably on a subnet not being used by the workstation's other network interfaces, and configure the MMX to use that IP address.
2. Choose a VPI:VCI pair (i, j) on which the MMX will transmit and receive its IP packets, and configure the MMX with this VPI:VCI.
3. Choose a possibly different VPI:VCI pair (i', j') on which the workstation will transmit and receive its IP packets, and create a Classical IP virtual interface on the workstation that associates the IP address chosen in Step 1 with VPI:VCI (i', j').
4. Set up a PVC in the ATM network so that cells originating from the MMX's switch port on VPI:VCI (i, j) are routed to the workstation's switch port on VPI:VCI (i', j'), and vice-versa.

Note that a PVC *must* be set up on the switch in Step 4, both when the MMX and workstation are using different ports of the ATM switch (as in Figure 3a), and when they are sharing a single port (Figure 3b). This is because the MMX only monitors cells coming from the switch; cells coming from the workstation are simply forwarded to the switch without any additional processing. If the MMX and workstation are sharing a switch port, the VPI:VCI pairs (i, j) and (i', j') must be unique. Although Figure 3a shows both devices attached to the same switch, the MMX and workstation are not required to be connected to the same switch if they are not sharing a switch port.

Details of how to perform Steps 1 and 2 can be found in the MMX Installation and Operation Manual. The documentation for the workstation's ATM NIC should describe how to create a Classical IP interface with a manually configured VPI and VCI, as required in Step 3. The procedure for creating a PVC as described in Step 4 should be detailed in the documentation of the ATM switch's network management interface.

If the above steps have been performed correctly, the workstation should be able to `ping` the MMX using the IP address chosen in Step 1. It should also be possible to `telnet` to the MMX, making it possible to configure the MMX using the same menus as those that are available on the serial port.

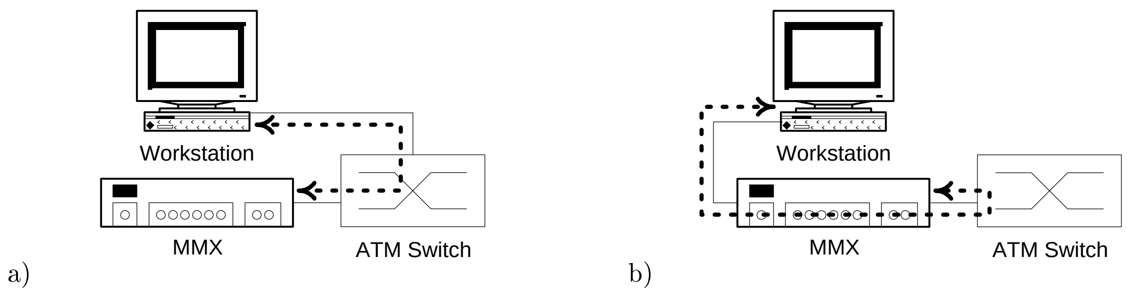

Figure 3: Two configurations of IP-over-ATM between a workstation and MMX. Solid lines indicate physical fiber links; dashed lines indicate the flow of cells carrying IP traffic. a) (*left*) The workstation and MMX are on different switch ports; cells from the workstation flow through the switch to the MMX, and vice-versa. b) (*right*) The MMX and workstation share a switch port; cells are still required to flow through the switch, though they will be routed out the same port on which they arrived.

3.3 One Workstation Controlling Multiple MMXs

Although an MMX can only be controlled by a single workstation at a time, a single workstation can simultaneously control more than one MMX. The number of MMXs that can be controlled by a workstation is limited only by the number of simultaneous Classical IP virtual interfaces supported by the workstation's NIC and/or NIC drivers.

Figure 4 shows an example of such a configuration. Workstation W_1 is simultaneously maintaining a Classical IP link with MMXs M_2, M_3 and M_4. Although not shown in Figure 4, W_1 could also connect to its own MMX if configured similarly to Figure 3a or 3b. Note that M_2 *cannot* maintain a direct connection to W_2, nor can M_3 connect to W_3, while they are connected W_1. However, if an application absolutely requires that an MMX simultaneously exchange IP packets with more than one workstation, one of the workstations can be configured as a router as described in the next section.

3.4 Using a Workstation as a Router

The MMX itself is only capable of supporting a single IP link to a single workstation at a time. However, by using the directly connected workstation as an IP router, it is often possible to allow multiple workstations simultaneous access to the MMX's SNMP management functions.

Although such a configuration is possible, it is not recommended. Many of the operations performed by MMXSNMP are not atomic; they require several SNMP commands to be sent and received to and from the MMX. Because the MMX does not provide any way of granting exclusive write access to a single entity, MMXSNMP must simply assume that it has exclusive access to the MMX's configuration when attempting to modify it. If more than one application is simultaneously sending and receiving SNMP commands to the MMX, the result is a race condition that may cause the MMX to be left in

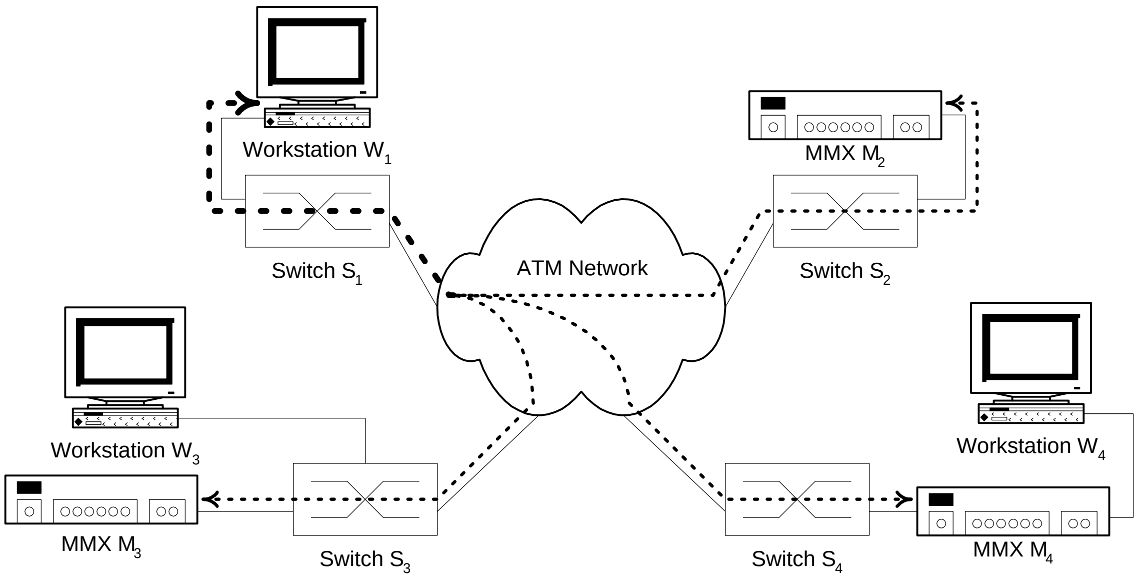

Figure 4: Workstation W_1 simultaneously controlling MMXs M_2, M_3 and M_4. Solid lines indicate physical fiber links; dashed lines indicate the flow of cells carrying IP traffic. M_2 is a stand-alone unit. M_3 is co-located with W_3, although M_3 and W_3 are not exchanging IP packets. M_4 and W_4 are also not exchanging IP packets, even though they are sharing a switch port.

an inconsistent state. If a workstation is configured as a router, granting SNMP access to many other workstations, the likelihood of such a race condition occurring greatly increases.

Most workstation operating systems are capable of performing the functions of an IP router. In general, the workstation must have at least one network interface in addition to its virtual Classical IP interface used to communicate with the MMX. The workstation should then be configured to forward packets between the MMX's Classical IP virtual interface and its other virtual or physical network interfaces. The MMX must also be configured to use the workstation as its Default Router.

For example, the simplest configuration might involve a workstation with both an ATM and Ethernet network interface. In Figure 5, Workstation W_1 has been configured to communicate with MMX M_1, similar to the workstation in Figure 3b. However, W_1 also has an Ethernet interface that provides connectivity to other workstations on its LAN, such as W_2. W_1 has also been configured to route packets between its two network interfaces. If W_2's routing table is configured to use W_1 as a router when trying reach M_1's IP address, packets generated at W_2 that are destined for M_1 will flow across the Ethernet to W_1, where W_1 will forward the packets to M_1 using its ATM interface.

More complex configurations are possible. For example, the router–workstation might be configured to forward packets between two virtual ATM interfaces. Figure 6 shows an all-ATM network with two workstations W_1 and W_2, each of which is connected to a single MMX, M_1 and M_2, respectively. W_1 and M_1 are configured as in Figure 3a; W_2 and M_2 are configured as in Figure 3b. The two workstations

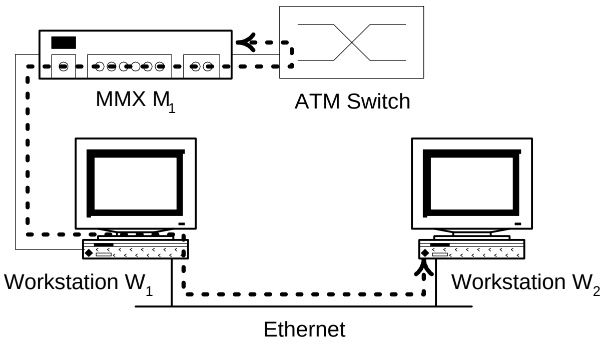

Figure 5: Workstation W_2 controlling MMX M_1 using Workstation W_1 as a router. Workstation W_2 sends packets to Workstation W_1, which then routes those packets to M_1 using its Classical IP interface as described in Section 3.2. Note that the configuration of W_1 and M_1 is the same as shown in Figure 3b.

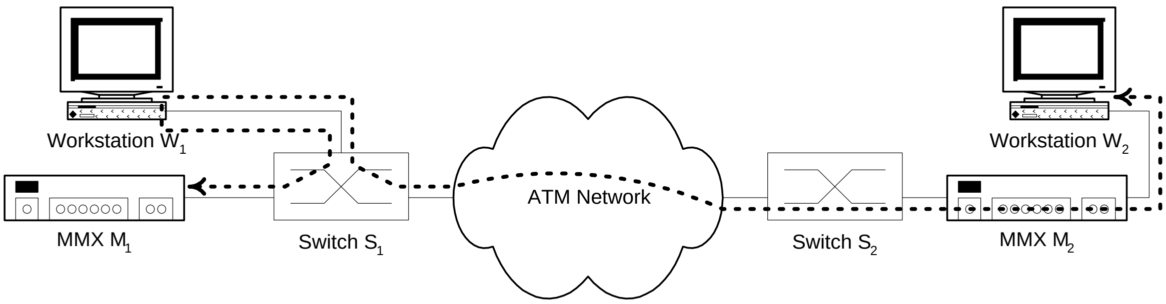

Figure 6: Workstation W_2 controlling MMX M_1 using Workstation W_1 as a router. W_2 sends packets to W_1, using any IP-over-ATM standard (such as Classical IP or LANE). W_1 then routes those packets to M_1 using its Classical IP interface as described in Section 3.2. Note that the configurations of W_1 and M_1 are the same as Figure 3a; W_2 and M_2 match Figure 3b.

have IP connectivity to each other through some IP-over-ATM method (for example, using LANE [9], MPOA [10], or another Classical IP virtual interface). If W_1 is configured as a router, W_2 can send IP packets to M_1 by routing them through W_1. Similarly, if W_2 is configured as a router, W_1 can route packets through W_2 to M_2, as shown in Figure 7.

Note in Figure 7 that IP packets from W_1 can not go directly to M_2 as was seen in Figure 4. In Figure 7, M_2 is logically connected to W_2; W_1 must therefore route its IP packets destined for M_2 through W_2, even though the packets physically pass through M_2 on their way to W_2.

Setting up workstations as routers is a complex topic; a full discussion of router configuration is well beyond the scope of this document. This section has glossed over many of the details involved in actually configuring a workstation as a router, such as the selection of a unique IP subnet dedicated to MMX IP addresses, subnet masks, and advertising routes with protocols such as RIP. Local experts and operating system documentation should be consulted for further detail on these topics.

Also, it should be reiterated that using a workstation as a router makes it much more likely that more than one user process will be trying to modify the configuration of an MMX at one time. The application modifying the MMX's configuration must use some method to ensure that only one process at a time is configuring the MMX, using lock files or some other form of interprocess communication. In the author's opinion, a better architecture for MMX control software is to designate a single workstation as the controller for each MMX—either one workstation controlling all MMXs, or each MMX being controlled by the workstation on the same desk. Some form of application-level IPC can then be used to request services from the controlling workstations, which will then configure the MMXs appropriately.

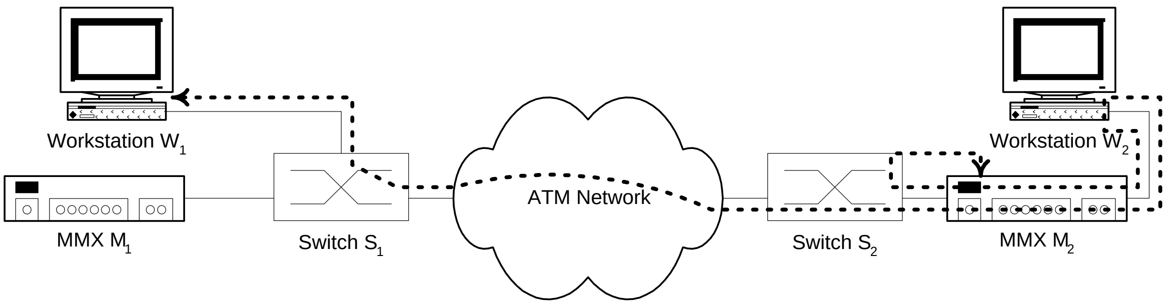

Figure 7: The configuration of this network is the same as in Figure 6; however, Workstation W_1 is now controlling MMX M_2 using Workstation W_2 as a router. Note that the cells carrying IP traffic must make an extra trip through Switch S_2 because W_2 is configured like the workstation in Figure 3b, and must "reflect" cells off of S_2 in order to reach M_2.

4 Using MMXSNMP For MMX Control

4.1 Introduction

This section gives a general overview of how to use the MMXSNMP library to write application software to control the MMX. MMXSNMP's major features are described, and demonstrated through the use of example programs. Most of the example programs included in this section are complete, working programs, which are available in electronic form with the rest of the MMXSNMP distribution.

This section is meant to be a tutorial, not a reference manual; many of the more esoteric features are not covered. The complete MMXSNMP reference manual can be found in Appendix B.

4.2 Compiling and Linking

Once MMXSNMP has been compiled and installed, as described in Section 1.4, there should be three files installed on your system that will be required to write applications using the library:

- `mmxsnmp.h` — The C++ header file that contains the declarations of the `mmxsnmp` and `vc` classes.
- `libmmxsnmp.a` — MMXSNMP's library of MMX-specific functions.
- `libsnmp.a` — The UCD SNMP library of low-level SNMP functions.

All C++ source files that wish to use the `mmxsnmp` class should `#include "mmxsnmp.h"`. Applications using class `mmxsnmp` should then be linked against both the `libmmxsnmp.a` and `libsnmp.a` libraries. The exact procedure for doing this depends on the particular compiler being used.

Note that other libraries may also be required, depending on the operating system. For example, Solaris 2.x requires that applications using Berkeley sockets explicitly be linked against the `socket` and `nsl` libraries.

4.2.1 Notes for Users of GCC

Parts of MMXSNMP are implemented using C++ derived exceptions. Unfortunately, this feature is not yet well-supported by many C++ implementations. MMXSNMP was developed using gcc 2.7.2.3, which includes limited but sufficient support for derived exceptions; it is described as "alpha–test code" in the documentation. Users of gcc 2.7.x should be aware that all code that uses MMXSNMP library calls, and MMXSNMP itself, must be compiled using the `-fhandle-exceptions` and `-frtti` compiler flags; also, source files compiled with these flags must have the statement `#include <typeinfo>` at the beginning of the file. Versions of gcc before 2.7.x may not work at all. gcc 2.8.0, when released, will have better exception support.

The example programs in this section all have the `#include <typeinfo>` statement, as required by gcc. g++ (gcc's C++ compiler) can be used to compile the example programs using a command such as:

```
g++ -fhandle-exceptions -frtti -o example example.cc -lmmxsnmp -lsnmp -lsocket -lnsl
```

The `-fhandle-exceptions` and `-frtti` flags are required for exception support; these flags should be used to compile all code that uses MMXSNMP. The `-lmmxsnmp` and `-lsnmp` flags tell the linker to link against the `libmmxsnmp.a` and `libsnmp.a` libraries, respectively. Note that for the `-l...` syntax to work correctly, the library files must be in a directory that the linker searches by default, such as `/usr/local/lib`. If the linker cannot find `libmmxsnmp.a` or `libsnmp.a`, either move those files into one of the standard directories, or use the `-L` option to specify an alternate location that should be searched for the files (for example, `-L/home/jelson/mmxsnmp`). Finally, note the `-lsocket` and `-lnsl` options, which cause the linking of libraries required for network programs under Solaris.

4.3 Your First MMXSNMP Program

MMXSNMP defines a class called `mmxsnmp`. Each instance of the class represents a control channel to a particular MMX. `mmxsnmp` has a constructor that takes a single argument: the Internet hostname of the MMX to be controlled (passed as a string; type `char *`). `mmxsnmp` has several dozen member functions, each of which can be used for checking on the current status of the MMX or changing its configuration in some way. All of these member functions are described in detail in the MMXSNMP Reference Manual, found in Appendix B.

Program 1 is an example of a simple program that uses MMXSNMP. Line 4 `#include`s the header file `mmxsnmp.h`, required in all programs that use MMXSNMP. Line 8 declares an instance of the class `mmxsnmp` called `my_mmx`; it controls the MMX reachable using the hostname specified in the constructor (`mmx1-atm.dcrt.nih.gov`). Line 10 prints a textual description of the MMX using `mmxsnmp`'s member function `sys_desc()`; this function is part of the System Group, which consists of functions for accessing general information about the MMX system. Line 11 uses another function in the System Group, `sys_serial_num()`, to print the serial number of the MMX.

When Program 1 is run, it gives the output:

Program 1 simple.cc: A simple program using MMXSNMP

```
#include <typeinfo>   // Required for GCC
#include <iostream.h>

#include "mmxsnmp.h"

int main(int argc, char *argv[])
{
  mmxsnmp my_mmx("mmx1-atm.dcrt.nih.gov");

  cout << my_mmx.sys_desc() << " is up!\n";
  cout << "Serial number is " << my_mmx.sys_serial_num() << "\n";
}
```

```
% simple
STS Technologies, MMX-04-10 is up!
Serial number is 001502
```

4.4 Specifying an SNMP Community String

The MMX is shipped with a default read–write community string of `private`. However, it is possible to change the community string for security purposes. This can only be done through the serial console or by using `telnet` to connect to the MMX's menu system; there is no way to change the community string using MMXSNMP. MMXSNMP will not be able to modify the configuration of the MMX unless the correct community string has been provided.

The community string can be specified by using a second form of `mmxsnmp`'s constructor that takes two arguments: the Internet hostname and the read–write community string. This is demonstrated in Program 2. On line 10, an instance of `mmxsnmp` is declared using the first form of the constructor, only specifying the desired hostname. Line 11 shows the second form, specifying both the desired hostname and read–write community string.

Also note that any number of instances of the `mmxsnmp` class can be declared in any part of the program; each one can represent a different MMX. In Program 2, two different MMXs are being controlled: `mmx1-atm.dcrt.nih.gov` and `mmx2-atm.dcrt.nih.gov`. Commands for both can be safely mixed.

As expected, the output of Program 2 is:

```
% community
STS Technologies, MMX-04-10, S/N 001502 up!
STS Technologies, MMX-04-10, S/N 001309 up!
```

4.5 Exception Handling

A big problem with Program 1 is that it makes no attempt to detect or handle run-time errors; in a real program, this would not be acceptable. MMXSNMP reports run-time errors by throwing exceptions. Programs can then use C++'s `try...catch` syntax, seen in lines 18–26 of Program 3, to test for errors and take some corrective action. If MMXSNMP detects an SNMP error, it is reported by throwing an

Program 2 community.cc: Specifying an SNMP community string

```
#include <typeinfo>   // Required for GCC
#include <iostream.h>

#include "mmxsnmp.h"

int main(int argc, char *argv[])
{
  // MMX1 uses the default community string, "private"; MMX2 uses a secret
  // community string
  mmxsnmp mmx1("mmx1-atm.dcrt.nih.gov");
  mmxsnmp mmx2("mmx2-atm.dcrt.nih.gov", "xyzzy");

  // Request information from MMX1
  cout << mmx1.sys_desc() << ", S/N " << mmx1.sys_serial_num() << " up!\n";

  // Request information from MMX2
  cout << mmx2.sys_desc() << ", S/N " << mmx2.sys_serial_num() << " up!\n";
}
```

instance of the exception class `snmp_exception`. If such an exception is caught, as shown in lines 23–26 of Program 3, `snmp_exception`'s member function `snmp_exception::debug()` can be used to describe the error on `stderr` and terminate the program.

Program 3 also improves on Program 1 by taking the Internet hostname of the MMX to be tested as a command-line argument. This lets us test different MMXs without recompiling the program, as would be required in Program 1. Line 16 checks to make sure an argument is provided; if so, Line 20 calls `test_mmx()`, passing the command-line argument to the testing function.

When testing an MMX that is properly configured, the exception-handling code is never called, as seen in this example:

```
% ping mmx1-atm
mmx1-atm.dcrt.nih.gov is alive
% exception mmx1-atm
Testing MMX at mmx1-atm...
STS Technologies, MMX-04-10, S/N 001502 up!
```

We can see the exception-handling code in action by trying to contact an MMX that is unreachable:

```
% ping mmx2-atm
no answer from mmx2-atm.dcrt.nih.gov
% exception mmx2-atm
Testing MMX at mmx2-atm...
SNMP: timeout waiting for response to .1.3.6.1.2.1.1.1.0
Unhandled SNMP exception
```

The last two lines of the output are error messages generated by `snmp_exception::debug()` when it is called on line 24 of Program 3. The first line describes the error as a timeout error, meaning that a response to an SNMP request was not received within 15 seconds. The number `.1.3.6.1.2.1.1.1.0` is the MIB object identifier of the SNMP request that failed; a list of these identifiers can be found in Appendix B of the MMX Installation and Operation Manual [3]. Finally, we see the message `Unhandled`

Program 3 exception.cc: Using MMXSNMP exception handling

```
#include <typeinfo>   // Required for GCC
#include <iostream.h>

#include "mmxsnmp.h"

void test_mmx(char *mmx_hostname)
{
  mmxsnmp mmx(mmx_hostname);

  cout << mmx.sys_desc() << ", S/N " << mmx.sys_serial_num() << " up!\n";
}

int main(int argc, char *argv[])
{
  if (argc == 1) {
    cerr << "usage: " << argv[0] << " mmx-hostname\n";
  } else {
    try {
      cout << "Testing MMX at " << argv[1] << "...\n";
      test_mmx(argv[1]);
      return 0;
    }
    catch (snmp_exception &e) {
      e.debug();
      // Never reached; e.debug() calls exit(-1)
    }
  }
}
```

`SNMP exception`, which is printed by `snmp_exception::debug()` just before it calls `exit(-1)` to terminate the program. Programs that require more robust error handling should not call `debug()`, but instead take some other action to recover from the failure of the library to execute the requested command.

Note that the SNMP timeout error illustrated above only indicates that MMXSNMP was not able to contact an SNMP agent running on the requested host (in this case, `mmx2-atm`). This can be caused by any one of a variety of reasons; for example, the MMX may be powered down, or IP connectivity to the MMX may not be configured properly (as described in Section 3). In addition, the same timeout error is generated when MMXSNMP is asked to contact a host that does not have an SNMP agent—even if IP connectivity to the host is working properly. For example:

```
% ping localhost
localhost is alive
% exception localhost
Testing MMX at localhost...
SNMP: timeout waiting for response to .1.3.6.1.2.1.1.1.0
Unhandled SNMP exception
```

In this case, IP connectivity to `localhost` is working correctly (unlike in the example with `mmx2-atm`, where even the `ping` failed); the timeout error is generated because `localhost`, while operating properly, has no SNMP agent that is listening to (or responding to) the SNMP requests generated by MMXSNMP.

`mmxsnmp`'s constructor, `mmxsnmp::mmxsnmp()`, can also throw an exception if there is a problem in initializing the UCD SNMP library. The most common example of this is attempting to establish a control session using an invalid hostname:

```
% exception ff
Testing MMX at ff...
unknown host: ff
Unhandled SNMP exception: SNMP library unavailable!
```

In this case, the first error message (`unknown host: ff`) is generated by the underlying UCD SNMP library. The next line, `Unhandled SNMP exception: SNMP library unavailable!`, is generated by the call to `snmp_exception::debug()` in line 24 of Program 3. This illustrates a shortcoming of MMXSNMP: because the underlying UCD SNMP library is written in C, some errors generated by that library print their own error messages or call `exit()`, without throwing an exception. Where possible, an error reported by the UCD SNMP library in the form of a function's return value causes an exception to be thrown by MMXSNMP.

One last type of exception, rarely thrown by MMXSNMP, is one that shows an SNMP agent was contacted successfully on a remote host, but the agent returned an error indicating that we sent an SNMP request for an unknown object. If this error is seen, it usually indicates a bug in MMXSNMP and should be reported to the author. However, we can force the error to occur by using our `exception` program to query a workstation that is running an SNMP agent:

```
% exception imacs
Testing MMX at imacs...
SNMP: object .1.3.6.1.2.1.1.1.0 not found!
Unhandled SNMP exception
```

In this case, the SNMP agent on the host `imacs` is receiving a request for an SNMP object in the MMX MIB. The agent on `imacs` is not an MMX and has no knowledge of any of the MMX's SNMP objects, so it returns an SNMP error, which MMXSNMP reports as an "object not found" exception.

4.6 The `vc` Class

Before discussing how to control audio and video streams, we must first describe the class `vc`, which is a simple container class for storing information about a particular "virtual circuit," or communication stream. `vc` is used as the argument or return value in several of `mmxsnmp`'s member functions. It contains three pieces of information:

- vpi — The ATM Virtual Path Identifier (VPI) used for the stream, in the range $0 \leq vpi \leq 127$.
- vci — The ATM Virtual Channel Identifier (VCI) used for the stream, in the range $0 \leq vci \leq 255$.
- dir — The direction of the stream; one of the constants `TRANSMIT`, `RECEIVE`, or `BIDIR`. These constants are defined in the enum `dir_type`.

Program 4 shows examples of using the class `vc`. The functions `vc::vci(int)`, `vc::vpi(int)`, and `vc::dir(dir_type)` can be used to set the data in an instance of `vc`, as seen on lines 11–13. Similarly, the accessor functions `vc::vpi()`, `vc::vci()`, and `vc::dir()` can be used to retrieve the data from `vc`, as seen on line 17. Values specified outside of the legal range for the `vpi` and `vci` will be silently rounded to the nearest legal value.

Program 4 vctest.cc: Examples of using the class `vc`

```
1    #include <typeinfo>   // Required for GCC
2    #include <iostream.h>
3
4    #include "mmxsnmp.h"
5
6    int main(int argc, char *argv[])
7    {
8      vc a(0, 250, TRANSMIT);
9      vc b, c, d;
10
11     b.vpi(10);
12     b.vci(230);
13     b.dir(RECEIVE);
14
15     c = vc(0, 250, TRANSMIT);
16
17     cout << "VPI and VCI of b are " << b.vpi() << ":" << b.vci() << "\n";
18
19     if (a == c)
20       cout << a << " == " << c << "\n";
21
22     if (b != c)
23       cout << b << " != " << c << "\n";
24
25     if ((a == b) || (b == c) || (a != c))
26       cout << "Something's wrong!\n";
27
28     // idle_vc is a constant belonging to mmxsnmp.  Since d was not initialized
29     // explicitly when declared on line 9, it defaults to the value of idle_vc.
30     if (d == mmxsnmp::idle_vc)
31       cout << "d is idle\n";
32   }
```

`vc` has a constructor, `vc::vc(int vpi, int vci, dir_type dir)`, that can be used to specify the values of `vpi`, `vci`, and `dir` at the time the instance is declared, as seen on line 8 of Program 4. `vc` also has a constructor that takes no arguments and initializes the instance to the value of the constant `mmxsnmp::idle_vc` (seen used on line 30). This constant is returned by some of `mmxsnmp`'s member functions to indicate an idle transmit or receive stream.

For convenience, `vc` also overloads the `==` and `!=` operators so that instances of `vc` may be compared to each other, as seen on lines 19, 22, 25, and 30. Two instances of `vc` are considered equal if and only if their respective `vpi`, `vci`, and `dir` values are all equal. `vc` also provides a function for performing stream output using the `<<` operator, used on lines 20 and 23 of Program 4.

The output of Program 4 is:

```
% vctest
VPI and VCI of b are 10:230
(0:250.T) == (0:250.T)
(10:230.R) != (0:250.T)
d is idle
```

4.7 Controlling Audio

4.7.1 Overview

MMXSNMP can be used to start and stop audio streams, configure various audio parameters, and check on the current status of the audio codec. For audio streams to actually be sent and received across an ATM network, PVCs must be created in the network to transit the cells from their source to their destination. PVC creation cannot be done with MMXSNMP; for more information, see Section 2.2.

The audio codec of the MMX has two operating modes, known simply as "Mode 0" and "Mode 1". Mode 0 is the simpler operating mode, capable of transmitting and receiving only one audio stream at a time. Mode 1 is more complex; although only one audio stream can be transmitted, it allows many audio streams to be simultaneously received, and automatically selects up to two of them to be played on the audio output (depending on which streams have active audio and have the highest priority). This section will only discuss the basics of MMX audio control, including basic audio parameters and how to control Mode 0 audio streams. Mode 1 is discussed in Section 5.

All MMXs participating in a conference must be using the same audio mode. This must be ensured by application software; the MMXs can not automatically synchronize themselves. The function `audio_mode(audio_mode_type mode)` can be used to switch the audio codec between the two modes described above. The `mode` argument must be the constant `MODE0` or `MODE1`. When the audio mode is switched from `MODE1` to `MODE0`, all open audio connections are closed. The companion function `audio_mode()` can be used to check the current operating mode of the audio codec.

Transmitted audio streams have one main parameter associated with them: the rate at which the analog audio is sampled by the audio codec. The sampling rate can be changed using the function `audio_sample_rate(audio_sample_type)`. The current sampling rate can be queried using the function `audio_sample_rate()`. Note that these functions follow the naming paradigm used by MMXSNMP: the functions for checking and setting parameters typically have the same name but take different arguments, similar to the `vc` class described in Section 4.6. Legal sample rates (the enum `audio_sample_type`) are `samp44K` (44.1 kHz), `samp29K` (29.4 kHz), `samp22K` (22.05 kHz), `samp17K` (17.64 kHz), `samp14K` (14.7 kHz), `samp11K` (11.025 kHz), `samp8K` (8.82 kHz), and `samp7K` (7.35 kHz).

Received audio streams have various volume parameters associated with them, including the master volume, the feedback volume, and the audio mix table. These parameters are discussed in Section 4.7.3. Received audio streams do not have a sampling rate parameter; the MMX can detect the sampling rate of incoming streams.

4.7.2 Starting and Stopping Audio

The function `audio_start(vc new_audio)` is used to start a new transmit or receive audio stream. Its sole parameter, `new_audio`, is an instance of the `vc` class described in Section 4.6. The new stream will use VPI `new_audio.vpi()` and VCI `new_audio.vci()`. `new_audio.dir()` should be set to either `TRANSMIT` or `RECEIVE`, depending on the type of stream desired. Setting `new_audio.dir()` to `BIDIR` is illegal in this context. If an audio stream in the direction of `new_audio.dir()` is already open, and the audio codec is operating in Mode 0, it is automatically closed before the new stream is opened. (We'll see in Section 5 that multiple audio receive streams are allowed in Mode 1.)

`audio_stop(dir_type dir)` is used to stop the transmission or reception of an audio stream. If `dir` is `TRANSMIT` and an audio transmit stream is active, the stream is closed. If `dir` is `RECEIVE` and an audio receive stream is active, the stream is closed. If `dir` is `BIDIR`, both of these operations are performed.

Program 5 audio.cc: Activating, deactivating, and checking the status of audio streams

```
1    #include <typeinfo>   // Required for GCC
2    #include <iostream.h>
3
4    #include "mmxsnmp.h"
5
6    void print_audio_status(mmxsnmp &mmx)
7    {
8      vc stream;
9
10     if ((stream = mmx.audio_status(TRANSMIT)) == mmxsnmp::idle_vc)
11       cout << "Not transmitting audio\n";
12     else
13       cout << "Transmitting on " << stream << '\n';
14
15     if ((stream = mmx.audio_status(RECEIVE)) == mmxsnmp::idle_vc)
16       cout << "Not receiving audio\n";
17     else
18       cout << "Receiving on " << stream << '\n';
19
20     cout << '\n';
21   }
22
23   main()
24   {
25      mmxsnmp mmx1("mmx1-atm.dcrt.nih.gov");
26
27      print_audio_status(mmx1);
28
29      mmx1.audio_start(vc(0, 100, TRANSMIT)); // Transmit on VPI 0, VCI 100
30      mmx1.audio_start(vc(0, 200, RECEIVE));  // Receive on VPI 0, VCI 200
31
32      print_audio_status(mmx1);
33
34      mmx1.audio_stop(BIDIR);
35
36      print_audio_status(mmx1);
37   }
```

`audio_status(dir_type dir)` is used to report the VPI and VCI of active audio streams. If `dir` is `TRANSMIT` and there is an active audio transmit stream, its VPI and VCI are returned via an instance of the class `vc`, described in Section 4.6. If no audio transmit stream is active, the constant `mmxsnmp::idle_vc` is returned instead. Similarly, if `dir` is `RECEIVE`, an instance of `vc` will be returned which will either contain the VPI and VCI of an active audio receive stream or be the constant `mmxsnmp::idle_vc`. Passing `BIDIR` to `audio_status()` is not legal.

The use of these functions is demonstrated in Program 5. The `audio_start()` function is used on line 29 to start transmitting audio on VPI 0, VCI 100. The function is used again on line 30 to start receiving audio on VPI 0, VCI 200. Also, note that on lines 29 and 30, the argument to `audio_start()` is an instance of the class `vc` created "on-the-fly" by explicitly passing the three required arguments to `vc`'s constructor.

Line 34 uses the `audio_stop()` function to stop both the transmission and reception of audio. Calling `audio_stop(BIDIR)` is no different than individually calling both `mmx1.audio_stop(TRANSMIT)` and `mmx1.audio_stop(RECEIVE)`.

The function `print_audio_status()`, defined on lines 6–21 of Program 5, reports active audio transmit and receive streams using the `audio_status()` function. First, it checks the return value of a call to `audio_status(TRANSMIT)`; if `mmxsnmp::idle_vc` is returned, it prints a message indicating that we are not transmitting audio. Otherwise, it outputs the stream on which audio is being transmitted. The same procedure is then repeated to check for an active audio receive stream.

The output of Program 5 is:

```
Transmitting on (0:120.T)    Call to print_audio_status() from line 27
Receiving on (0:140.R)

Transmitting on (0:100.T)    Call to print_audio_status() from line 32
Receiving on (0:200.R)

Not transmitting audio       Call to print_audio_status() from line 36
Not receiving audio
```

The first call to `print_audio_status()` comes at the beginning of Program 5, on line 27—before we have attempted to turn any audio streams on or off. In our example, the MMX happens to have been previously been configured to transmit and receive audio on VCIs 120 and 140, respectively, as is reported in the first two lines of output. The next call, on line 32 of Program 5, comes after the calls to `audio_start()`; the following two lines of output verify that we are indeed transmitting audio on VCI 100 and receiving on VCI 200. The final call comes after the call to `audio_stop()`, verifying that both the audio streams have been terminated.

4.7.3 Audio Volume Parameters

Audio receive streams have various volume parameters with them. These parameters are:

- The *feedback volume*. This is the percentage of the local audio input which is replayed back to the local audio output. If the user is wearing headphones instead of using speakers as the local audio output (to reduce audio feedback noise), the user cannot hear his or her own voice while speaking; this can be distracting. Turning on the feedback volume remedies this by replaying the audio input to the audio output in the same way that a telephone does.
- The *mix table*. Each audio stream is transmitted on two channels (A and B), and two audio output channels are available (also named A and B). The audio mix table defines how much of each received audio channel should be routed to each local audio output. This can be useful, for example, to route one site's audio to the user's left speaker, and one site to the right speaker.
- The *master volume*. This is like a master "volume knob" to increase or decrease the total audio volume after the feedback volume and mix table have both been applied.

The feedback volume is controlled with the function `audio_feedback_volume(int vol)`; the current feedback volume can be queried using `audio_feedback_volume()`. Similarly, the master volume can be set and queried using the `audio_master_volume(int vol)` and `audio_master_volume()` functions, respectively. The mix table specification is complex; see Sections 7.14 and 7.15 of Appendix B for details.

Each volume parameter described above is specified as an `int` ranging in the range of 0...100, representing the percentage of the maximum possible volume. 0 indicates the audio is off, and 100 indicates the audio is being played at its maximum volume.

4.8 Controlling Video

4.8.1 Overview

The MMX is capable of simultaneously sending a single video stream and receiving a single video stream. MMXSNMP can be used to start and stop the streams, configure various video parameters, and check on the current status of the video codec. For video streams to actually be sent and received across an ATM network, PVCs must be created in the network to transit the cells from their source to their destination. PVC creation cannot be done with MMXSNMP; for more information, see Section 2.2.

Each transmitted and received video stream has a parameter associated with it called the Q Factor. The Q Factor controls the JPEG compression and decompression algorithm applied to each frame of transmitted and received video. Higher Q Factors result in more compression, lower required bandwidth for the video stream, and greater degradation of the image. Lower Q Factors result in less compression, higher bandwidth required for the video stream, and less degradation of the image. The actual compression achieved and bandwidth required are dependent on other factors such as the complexity of the image and the amount of motion within the field of view. Legal values for the Q Factor are in the range $10 \leq Q \leq 1000$.

For a video stream to be properly received, it must be received using the same Q Factor with which the stream was transmitted. It is the responsibility of the application to ensure that the synchronization of Q Factors is maintained. Unlike the audio sampling rate, which can be selected on the transmitting side without coordinating with the receiver, the MMX *can not* automatically detect the Q Factor of an incoming video stream. Although a video stream must always be received using the same Q Factor on which it was transmitted, an MMX can potentially be transmitting and receiving video using different Q Factors. In other words, "asymmetrical" video streams are allowed; for example, it is legal for MMX M_1 to be transmitting video to MMX M_2 at $Q = 30$ and receiving at $Q = 40$, while M_2 is transmitting video to M_1 at $Q = 40$ and receiving at $Q = 30$.

Transmitted video streams have two additional parameters associated with them: the Video Input Source and PLL Mode. The Video Input Source simply selects which video input port is used for the transmitted video. The MMX has three analog video inputs—two composite inputs (labeled `CIN1` and `CIN2`), and one S-Video input (labeled `SIN1`). The PLL Mode is used to describe the type of video source connected to port specified by the Video Input Source; legal values are `CAMERA` and `VTR`. This parameter is required for proper synchronization between the video source and the MMX's video codec.

MMXSNMP always maintains a "current value" for each of the parameters described above, initialized to a particular default value. For each parameter, MMXSNMP has member functions for checking the current value of the parameter and changing it to a new value. Again, the functions for checking and setting the parameters typically have the same name but take different arguments, similar to the `vc` class described in Section 4.6. Whenever a new video stream is created, is uses the current value of all the parameters. When a parameter is changed, it immediately takes effect on all currently active streams, in addition to becoming the value used for future streams.

The Transmit Q Factor and Receive Q Factor both have a default of 30 and can be changed with the `video_q_factor()` function. The default Video Input Source is `SIN1` and can be changed using the `video_input()` function. The PLL Mode has a default of `CAMERA` and can be changed using the `video_pll_mode()` function.

4.8.2 Starting and Stopping Video

Starting and stopping video streams works the same way as starting and stopping Mode 0 audio streams. Most of the semantics of the `video_start()`, `video_stop()`, and `video_status()` functions are the same as the analogous audio control functions.

The function `video_start(vc new_video)` is used to start a new transmit or receive video stream. Its sole parameter, `new_video`, is an instance of the `vc` class described in Section 4.6. The new stream will use VPI `new_video.vpi()` and VCI `new_video.vci()`. `new_video.dir()` should be set to either `TRANSMIT` or `RECEIVE`, depending on the type of stream desired. Setting `new_video.dir()` to `BIDIR` is illegal in this context. If a video stream in the direction of `new_video.dir()` is already open, it is automatically closed before the new stream is opened.

If `new_video.dir()` is `TRANSMIT`, the MMX will begin transmitting video from the current Video Input Source, using the current Transmit Q Factor and PLL Mode. Similarly, if `new_video.dir()` is `RECEIVE`, the MMX will begin receiving video using the current Receive Q Factor.

`video_stop(dir_type dir)` is used to stop the transmission or reception of a video stream. If `dir` is `TRANSMIT` and a video transmit stream is active, the stream is closed. If `dir` is `RECEIVE` and a video receive stream is active, the stream is closed. If `dir` is `BIDIR`, both of the above operations are performed.

`video_status(dir_type dir)` is used to report the VPI and VCI of active video streams. If `dir` is `TRANSMIT` and there is an active video transmit stream, its VPI and VCI are returned via an instance of the class `vc`, described in Section 4.6. If no video transmit stream is active, the constant `mmxsnmp::idle_vc` is returned instead. Similarly, if `dir` is `RECEIVE`, an instance of `vc` will be returned which will either contain the VPI and VCI of an active video receive stream or be the constant `mmxsnmp::idle_vc`. Passing `BIDIR` to `video_status()` is not legal.

4.8.3 Changing the Q Factor

The function `video_q_factor(int Q, dir_type dir)` can be used to select new values for the Transmit and Receive Q Factors. The desired Q Factor (the parameter `Q`) must be in the range $10 \leq Q \leq 1000$; values specified outside of this range will be silently rounded to the nearest legal value. If `dir` is `TRANSMIT`, the Transmit Q Factor is set to `Q`. If `dir` is `RECEIVE`, the Receive Q Factor is set to `Q`. If `dir` is `BIDIR`, both the Transmit and Receive Q Factors are set to `Q`. The new Q Factor immediately takes effect on active video streams, as well as becoming the default Q Factor for future video streams.

The function `video_q_factor(dir_type dir)` is used to report the currently selected Q Factor. If `dir` is `TRANSMIT` and there is an active video transmit stream, the Q Factor of that stream is reported; if no video transmit stream is active, the Transmit Q Factor that will be used for future video transmit streams is reported. If `dir` is `RECEIVE`, the Receive Q Factor is reported in the same way. Setting `dir` to `BIDIR` is not allowed.

Program 6 shows an example how to set the Q Factor on video streams. It also demonstrates the technique described in Section 4.8.1 of setting up video streams with asymmetrical Q Factors. In lines 11–14, `mmx1` is configured to transmit video with a Q Factor of 30, and `mmx2` is configured to receive video with a Q Factor of 30. This assumes that a call exists in the ATM network so that cells originating at `mmx1` with a VPI:VCI of 0:100 are routed to `mmx2` with a VPI:VCI of 0:101.

In lines 17–20, the reciprocal video stream is configured; however, in this case, the transmitter (`mmx2`) and receiver (`mmx1`) are synchronized at a Q Factor of 75. Again, we assume that the appropriate PVC has been set up in the ATM network. Although the two video streams are using different Q Factors,

Program 6 qfactor.cc: Setting the Q Factor of video streams

```
1     #include <typeinfo>   // Required for GCC
2
3     #include "mmxsnmp.h"
4
5     main()
6     {
7        mmxsnmp mmx1("mmx1.dcrt.nih.gov");
8        mmxsnmp mmx2("mmx2.dcrt.nih.gov");
9
10       // Set up video from mmx1 to mmx2 using a Q Factor of 30
11       mmx1.video_q_factor(30, TRANSMIT);
12       mmx1.video_start(vc(0, 100, TRANSMIT)); // Transmit on VPI 0, VCI 100
13       mmx2.video_q_factor(30, RECEIVE);
14       mmx2.video_start(vc(0, 101, RECEIVE));  // Receive on VPI 0, VCI 101
15
16       // Set up video from mmx2 to mmx1 using a Q Factor of 75
17       mmx2.video_q_factor(75, TRANSMIT);
18       mmx2.video_start(vc(0, 200, TRANSMIT)); // Transmit on VPI 0, VCI 200
19       mmx1.video_q_factor(75, RECEIVE);
20       mmx1.video_start(vc(0, 201, RECEIVE));  // Receive on VPI 0, VCI 201
21
22       // ...some time later, we want to change the Q Factor in both directions
23       mmx1.video_q_factor(20, BIDIR);
24       mmx2.video_q_factor(20, BIDIR);
25    }
```

each receiver is synchronized with its corresponding transmitter.

On lines 23 and 24, `video_q_factor()` is again called to change the Transmit and Receive Q Factors to 20 on both MMXs. Note that in lines 11–20, the Q Factor is set *before* the streams are activated. However, in lines 23–24, the Q Factor is set to 20 while active streams already exist; this has the effect of changing both the currently active Q Factor and the Q Factor to be used for future streams. In other words, the active streams immediately start using Q Factor of 20, and new streams would also use a Q Factor of 20 (for example, if `video_start()` were called again after line 24).

5 Using the MMX in Conference Mode

This section is taken from the Draft Working Note on using the MMX in Conference Mode, written by STS Technologies, Inc. It is reprinted here by permission.

The MMX has two modes of audio. Mode "zero" is a mode that supports CD quality stereo sound (as well as lower audio bandwidths). Mode "one" is a conference mode, designed to support "voice queuing" needed in conference applications where the primary audio component is a voice. While mode "one" can support higher audio bandwidths, a typical application will have the audio bandwidth set in the range of 6 to 8 kHz. Mode "one" is monaural audio. The analog audio input can be switched to the "A" or the "B" input, or a mix of the two. The analog audio output is fed to both output ports. The gain to each output port can be set independently.

The necessary information to make the audio voice queuing work is a mixture of work by the audio board itself, the MMX on-board microprocessor, and information passed to the MMX by the application controlling it. This section is intended to be a high-level description of what that information is, where it is created, and the extent of the information distribution.

The audio voice queuing is designed so MMX conferences will mimic the dynamics of an in-person group conversation, where all conference participants are looking at the person talking. When another person speaks up, everyone looks at the new speaker, with the exception of the new speaker who is typically looking at the previous speaker Speaking etiquette that occurs in natural conversation is expected to carry over into electronic conferences.

The audio voice queuing described here is intended to be used in controlling conferences such that the person speaking will "get the floor." "Getting the floor" means that all participants in the conference will "see" and "hear" the person who has the floor. The person who has the floor will see the previous speaker. The audio control will allow two speakers to be heard by all the conference participants at the same time. Thus, in a two way conversation, all the people in the conference will always be watching the current speaker, the two people speaking will see each other, and all the conference participants will hear both people speaking—even when they both talk at once. There is no central control point involved; the control for this switching of the active video and audio at each MMX is controlled by the voice queuing circuits. There are no hand interactions, such as pushing buttons or clicking a computer mouse, required by the participants after the conference begins.

We will first describe the primitive functions of the MMX, then show how these primitives can be used to create a conference. The MMX audio section contains a high power DSP that continuously monitors the digital traffic being received by the MMX. There is a "global memory" in the MMX that is accessible by both the audio DSP, and by applications using MMXSNMP. Through this path, the DSP in each MMX can be loaded with information pertaining to the overall conference.

The MMX primitives are listed below. The corresponding function in the MMXSNMP library is listed at the end of each entry.

1. Each MMX has a source identification number (an ID). This Source ID is included in the payload of each audio cell transmitted by this MMX. (`audio_local_src_id()`)
2. Each MMX will receive an audio "threshold" value. Any analog audio input received by this MMX will be considered to be "zero" until the sound level exceeds the "threshold" value set for this MMX. This feature allows each MMX in a conference to be adjusted to ignore background noise. (`audio_vox_thresh()`)
3. Each MMX will receive a flag bit that determines if the audio stream with the local ID will be mixed in with the feed to the local analog audio output (`audio_feedback_volume()`
4. Each MMX will send to the audio outputs the mix of the two ATM digital audio streams with the lowest IDs. (Unless one of the streams contains local ID—then, *if a flag is set*, only the non-local ID incoming ATM digital audio stream will be sent to the audio output. This feature helps audio feedback problems.)
5. If an MMX is receiving its own digital audio stream, the volume of the total received audio sent to the audio output is controlled by a separate gain setting. This is another feature to help control audio feedback problems.
6. If the audio "threshold" is exceeded, then the MMX will begin transmitting ATM digital audio cells if the local ID value is one of the two lowest IDs in the set of incoming digital audio streams. Note that while each MMX has an ATM channel that, in general, no more than two will be transmitting data at any one time. It is possible that short bursts of time could have several MMXs transmitting audio ATM cells. (For example, a simultaneous laugh of all the conference participants in response to a joke.)
7. If an MMX is transmitting audio ATM cells and receives two IDs that are less than the local ID, the MMX stops transmitting audio ATM cells. (Settles out the all-laughing issue quickly and gives priority to certain speakers over others.)
8. When an MMX receives an audio cell with a new ID, the MMX will build up the volume gain over the first cell's audio samples from zero to the set gain. If the new ID is replacing another ID, then the last cell's audio samples from the other ID will reduce in gain from the set value to zero. (This "feathering" of the on/off helps the sound quality.)
9. If the MMX is transmitting audio ATM cells, and is receiving no other IDs, the MMX will keep transmitting regardless of "threshold" and audio input. This keeps some background noise in the conference, as dead silence is not pleasing. (Each user has the ability to turn off their microphone via another command for privacy.)

10. Each MMX will place the IDs of the audio streams being fed to the audio output in the local MMX's global memory. ID "zero" is reserved to indicate no audio cells. Thus, most of the time, only one of the two global memory locations reserved for IDs will have a non-zero value.
11. Once an MMX starts transmitting audio, it will continue to transmit for at least a user-configurable amount of time. The default is one fifth of a second. (`audio_vox_hold()`)
12. The user ID of "zero" can be used to set an MMX to a "listen only" mode. A MMX with its ID set to "zero" cannot transmit its audio.

Most of the time during a conference, there will be one ATM audio stream active in the ATM network. Once in a while there will be two ATM audio streams active for a short period as the conference speech flow moves from one person to another. The rest of the functions listed above are to cover the "boundary cases" that make the difference between an application that is just barely usable to one that works smoothly.

The two IDs placed on global memory by the audio DSP contain no notion of which ID is the last one to be added to the global memory. It is the responsibility of applications to "watch" the global memory and determine which video input to display at each MMX. It is also up to the higher level to determine if the local video should be transmitted. (In general, there will only be two active video ATM cell streams in the network at any one time.)

The network required to support the MMX conferences efficiently requires that the network be configured as a "many to many" network. Many-to-one and a multi-cast function will also work. Large conference support will require a switch with high-bandwidth ports or a connection scheduler that allows the scheduling of a large number of ports to a single common port, with the total bandwidth allocated to the conference at this one common port being on the order of two video data streams (plus a bit for audio streams). If the call scheduler does not include the ability to understand that the bandwidth at the common port (the "one" of the many-to-one connection) is limited to the bandwidth of only two of all the ports routed to this single port, then the call schedule will prohibit the setting up of the conference.

The "mode one" audio cell is made up of the ATM 5 byte header, followed by a 4 byte ID value, followed by 22 2-byte samples.

A References

[1] Jr. J.R. Cox, W.D. Richard, K. Krieger, and B. Gottlieb. The Washington University Multimedia System. *ACM Multimedia Systems*, 1:120–131, 1993.

[2] M. Schoffstall, M. Fedor, J. Davin, and J. Case. RFC 1157: A Simple Network Management Protocol (SNMP), May 1990. Available on the Internet at `ftp://ftp.internic.net/rfc/rfc1157.txt`.

[3] STS Technologies, Inc., 3800 McKelvey Road, Bridgeton, MO 63044. *MultiMedia eXplorer Installation and Operation Manual*, 1996.

[4] STS Technologies, Inc., 3800 McKelvey Road, Bridgeton, MO 63044. *MMX030 MIB*, 1996.

[5] STS Technologies, Inc. Draft working note. Draft of a manual describing the use of the audio codec's Mode 1., Oct 1996.

[6] ATM Forum Technical Committee. ATM User–Network Interface Specification Version 3.0. Specification `af-uni-0010.001`, ATM Forum, Mountain View, CA, September 1993. Available on the Internet at `ftp://ftp.atmforum.com/pub/approved-specs/af-uni-0010.001.ps`.

[7] D. Ginsburg. *ATM Solutions for Enterprise Internetworking*. Addison–Wesley, 1996.

[8] M. Laubach. RFC 1577: Classical IP and ARP over ATM, January 1994. Available on the Internet at `ftp://ftp.internic.net/rfc/rfc1577.txt`.

[9] ATM Forum Technical Committee. LAN Emulation over ATM Version 1.0. Specification `af-lane-0021.000`, ATM Forum, Mountain View, CA, January 1995. Available on the Internet at `ftp://ftp.atmforum.com/pub/approved-specs/af-lane-0021.000.ps`.

[10] ATM Forum Technical Committee. Multi-Protocol Over ATM Specification v1.0. Specification `af-mpoa-0087.000`, ATM Forum, Mountain View, CA, July 1997. Available on the Internet at `ftp://ftp.atmforum.com/pub/approved-specs/af-mpoa-0087.000.ps`.

B Reference Manual

1 General Functions

1.1 **mmxsnmp::mmxsnmp** (char *hostname, char *community = "private")

Constructor

The constructor initializes the mmxsnmp class by initializing the underlying UCD SNMP library. If the UCD library reports some error initializing the library (for example, due to an invalid hostname), the constructor will throw an exception.

The second argument, `community`, has a default value of `private`—the default read-write community string shipped with the MMX. If the read-write community string of the desired MMX has been configured to be a different string, it must be passed as the second argument of the constructor.

Parameters:

`hostname` — The Internet hostname of the MMX to be controlled. This can be specified as either an alphabetic hostname or an IP number in dotted-quad format.
`community` — (Optional) The read-write community string of the MMX to be controlled. If not specified, it defaults to `private`.

1.2 **mmxsnmp::˜mmxsnmp** ()

Destructor

The destructor is normally not called explicitly. It calls the termination routines of the underlying UCD SNMP library.

2 STS System Functions

This group includes functions that provide information about the MMX system as a whole. The pointers returned are pointers into private buffers belonging to the library.

2.1

char* **mmxsnmp::sys_desc** ()

Return Value: The manufacturer, name, and model number of the system.

2.2

char* **mmxsnmp::sys_serial_num** ()

Return Value: The serial number of the system.

2.3

char* **mmxsnmp::sys_hw_addr** ()

Return Value: The hardware address of the system.

2.4

char* **mmxsnmp::sys_serviced** ()

Return Value: The date that the system was last serviced. If this date is unknown, the value of this string is "`WED DEC 31 23:59:59 1969 GMT`".

2.5

void **mmxsnmp::sys_reset** ()

Calling this function causes a hard reset of the MMX.

3 Hardware System Information

This group provides general information about the hardware components that comprise the MMX030 system. This information will be useful for service and system maintenance, and may also be used by management software to maintain compatibility across software versions. The pointers returned are pointers into private buffers belonging to the library.

3.1

char* **mmxsnmp::hw_cpu_serial_num** ()

Return Value: The ATMizer module serial number.

3.2

char* **mmxsnmp::hw_cpu_revision** ()

Return Value: The ATMizer module revision number.

3.3

char* **mmxsnmp::hw_net_if_serial_num** ()

Return Value: The network interface module serial number.

3.4

char* **mmxsnmp::hw_net_if_revision** ()

Return Value: The network interface module revision number.

3.5

char* **mmxsnmp::hw_audio_serial_num** ()

Return Value: The audio codec module serial number.

3.6

char* **mmxsnmp::hw_audio_revision** ()

Return Value: The audio codec module revision number.

3.7

char* **mmxsnmp::hw_video_serial_num** ()

Return Value: The video codec module serial number.

3.8

char* **mmxsnmp::hw_video_revision** ()

Return Value: The video codec module revision number.

4 Software System Information

This group provides general information about the software components that comprise the MMX030 system. This information will be useful for service and system maintenance, and may also be used by management software to maintain compatibility across software versions. The pointers returned are pointers into private buffers belonging to the library.

4.1 char* **mmxsnmp::sw_os** ()

Return Value: The name of the OS Kernel running on the system.

4.2 char* **mmxsnmp::sw_os_revision** ()

Return Value: The revision of the OS Kernel running on the system.

4.3 char* **mmxsnmp::sw_os_date** ()

Return Value: The build date of the OS Kernel running on the system.

4.4 char* **mmxsnmp::sw_audio_os** ()

Return Value: The name of the OS Kernel running on the Audio Codec.

4.5

char* **mmxsnmp::sw_audio_os_revision** ()

Return Value: The revision of the OS Kernel running on the Audio Codec.

4.6

char* **mmxsnmp::sw_audio_os_date** ()

Return Value: The build date of the OS Kernel running on the Audio Codec, in the format YYMMDD.

5 System Configuration

These functions provide access to system-wide configuration parameters.

5.1 baud_type **mmxsnmp::conf_baud** ()

Reports the current console baud rate

Reports the current baud rate of the system console port. Legal baud rates are `baud300`, `baud600`, `baud1200`, `baud2400`, `baud4800`, `baud9600`, and `UNKNOWNBAUD`.

Return Value: The current baud rate.

5.2 void **mmxsnmp::conf_baud** (baud_type b)

Sets the console baud rate

Sets the baud rate of the system console port. Legal baud rates are `baud300`, `baud600`, `baud1200`, `baud2400`, `baud4800`, and `baud9600`.

Parameters: `b` — The desired baud rate.

5.3

int **mmxsnmp::conf_kill_char** ()

Reports the current console line kill character

Return Value: The ASCII line kill character currently used on the system console port.

5.4

void **mmxsnmp::conf_kill_char** (int c)

Sets the console line kill character

Sets the ASCII line kill character used on the system console port.

Parameters: `c` — The desired line kill character.

5.5

int **mmxsnmp::conf_bkspc_char** ()

Reports the current console backspace character

Return Value: The ASCII backspace character currently used on the system console port.

5.6

void **mmxsnmp::conf_bkspc_char** (int c)

Sets the console backspace character

Sets the ASCII backspace character used on the system console port.

Parameters: `c` — The desired backspace character.

6 Network Interface Information

These functions provide statistics for the ATM network interfaces on the system.

6.1

int **mmxsnmp::atm_tx_cells** ()

Return Value: Count of cells transmitted from the system.

6.2

int **mmxsnmp::atm_rx_cells** ()

Return Value: Count of cells received by the system.

6.3

int **mmxsnmp::atm_hec_errors** ()

Return Value: Count of cells received with uncorrectable HEC errors.

6.4

int **mmxsnmp::atm_path_bip_errors** ()

Return Value: Count of Path BIP errors.

6.5

int **mmxsnmp::atm_path_febe_errors** ()

Return Value: Count of Path FEBE errors.

6.6

int **mmxsnmp::atm_line_bip_errors** ()

Return Value: Count of Line BIP errors.

6.7

int **mmxsnmp::atm_line_febe_errors** ()

Return Value: Count of Line FEBE errors.

6.8

int **mmxsnmp::atm_sect_bip_errors** ()

Return Value: Count of Section BIP errors.

7 Audio Functions

These functions provide information and control of the audio codec subsystem of the MMX. The audio codec must always be operating in one of two modes: `MODE0` or `MODE1`. In `MODE0`, a single continuous audio data stream is generated and processed by the audio codec. In `MODE1`, a single audio stream is generated only when the input level is above the input threshold, and multiple audio streams may be simultaneously received. All audio codecs must be set to the same mode to communicate.

7.1 audio_mode_type **mmxsnmp::audio_mode** ()

Reports the current mode of the audio codec

Valid modes are `MODE0` and `MODE1`.

Return Value: The current operating mode.
See Also: audio_mode(audio_mode_type mode)

7.2 void **mmxsnmp::audio_mode** (audio_mode_type mode)

Sets the mode of the audio codec

Valid modes are `MODE0` and `MODE1`. Upon switching from `MODE1` to `MODE0`, all open audio connections will be closed.

Parameters: `mode` — The desired operating mode: `MODE0` or `MODE1`.
See Also: audio_mode()

7.3

void **mmxsnmp::audio_start** (vc new_audio)

Starts transmission or reception of an audio stream

If `new_audio.dir()` is `TRANSMIT`, the audio codec will begin transmitting audio on VPI `new_audio.vpi()` and VCI `new_audio.vci()`. If another audio transmit stream is already active, that stream will be automatically closed.

If `new_audio.dir()` is `RECEIVE` and the audio codec is operating in `MODE0`, the audio codec will begin receiving audio on VPI `new_audio.vpi()` and VCI `new_audio.vci()`. If another audio receive stream is already active, that stream will be automatically closed.

If `new_audio.dir()` is `RECEIVE` and the audio codec is operating in `MODE1`, and the number of active audio receive streams is less than `mmxsnmp::MAX_AUD_RECV`, the audio codec will begin receiving audio on VPI `new_audio.vpi()` and VCI `new_audio.vci()`, while keeping open any other audio receive streams that may already be active. If `mmxsnmp::MAX_AUD_RECV` audio receive streams are already open, the new stream will not be created.

Parameters: `new_audio` — An instance of class `vc` containing the desired VPI, VCI and direction of the new audio stream. `new_audio.dir()` must be `TRANSMIT` or `RECEIVE`.

See Also: audio_stop(dir_type dir)
audio_stop(vc audio)

7.4

void **mmxsnmp::audio_stop** (dir_type dir)

Stops audio from being transmitted and/or received

If `dir` is `TRANSMIT` and an audio transmit stream is active, the stream is closed. If `dir` is `RECEIVE` and at least one audio receive stream is active, all audio receive streams are closed. If `dir` is `BIDIR`, both of the above operations are performed.

Note that if the audio codec is operating in `MODE1` and more than one audio receive stream is active, this function cannot be used to selectively close a subset of those streams; use `audio_stop(vc audio)` instead.

Parameters: `dir` — The direction of the audio stream to be closed: `TRANSMIT`, `RECEIVE`, or `BIDIR`.

See Also: audio_start(vc new_audio)
audio_stop(vc audio)
audio_status(dir_type dir)

7.5

void **mmxsnmp::audio_stop** (vc audio)

Closes a specific audio stream

If there is an active audio stream with VPI `audio.vpi()` and VCI `audio.vci()`, in direction `audio.dir()`, that stream is closed. `audio.dir()` should be either `TRANSMIT` or `RECEIVE`. If no such stream is active, this function has no effect.

This function is primarily useful if the audio codec is operating in `MODE1` and the caller wishes to close one audio receive stream when more than one is active. To close all audio receive streams (or the audio transmit stream), `audio_close(dir_type dir)` can be used instead. Multiple calls to this function are required if the caller wishes to close more than one, but not all, active audio receive streams.

Parameters: `audio` — An instance of class `vc` containing the desired VPI, VCI and direction of audio stream to be closed. `audio.dir()` must be `TRANSMIT` or `RECEIVE`.

See Also: audio_stop(dir_type dir)
audio_recv_status(vc audio[], int &count)

7.6

vc **mmxsnmp::audio_status** (dir_type dir)

Reports active audio streams

Reports the VPI and VCI (through an instance of class `vc`) of active audio streams.

If `dir` is `TRANSMIT` and there is an active audio transmit stream, it is returned. If no audio transmit stream is active, the constant `mmxsnmp::idle_vc` is returned.

If the audio codec is operating in `MODE0`, `dir` is `RECEIVE`, and there is an active audio receive stream, it is returned. If no audio receive stream is active, the constant `mmxsnmp::idle_vc` is returned.

This function cannot be used to check the status of active `MODE1` audio receive streams because more than one stream can potentially be active; use `audio_recv_status(vc audio[], int &count)` instead. If the audio codec is operating in `MODE1` and `dir` is `RECEIVE`, this function will always return the constant `mmxsnmp::idle_vc`.

Return Value:	The active audio stream or the constant `mmxsnmp::idle_vc`.
Parameters:	`dir` — `TRANSMIT` or `RECEIVE`.
See Also:	audio_start(vc new_video) audio_stop(dir_type dir) audio_stop(vc audio) audio_recv_status(vc audio[], int &count)

7.7

void **mmxsnmp::audio_recv_status** (vc audio[], int &count)

Reports active audio receive streams

This function is primarily used to report all active audio receive streams when the audio codec is operating in `MODE1`; `audio_status(dir_type dir)` is a simpler way to check the status of audio transmit streams or a single `MODE0` audio receive stream.

The caller provides the array `audio` in which the function can store instances of the class `vc`, each representing a single active audio receive stream. `count` should contain the size of the array (number of elements); this determines the maximum number of streams that can be reported. To ensure that all streams are reported, the size of `audio` should be ≥ `mmxsnmp::MAX_AUD_RECV`.

When called, the function modifies the argument `count` in-place to report the number of active audio receive streams. The streams are returned in array positions `audio[0]` ... `audio[count-1]`.

Parameters: `audio` — An array in which the active audio receive streams are stored.
`count` — The size (number of elements) of the array `audio`.

7.8

audio_sample_type **mmxsnmp::audio_sample_rate** ()

Reports the current audio sampling rate

Reports the current rate at which analog audio signals are processed by the audio codec. Legal sampling rates are `samp44K` (44.1 kHz), `samp29K` (29.4 kHz), `samp22K` (22.05 kHz), `samp17K` (17.64 kHz), `samp14K` (14.7 kHz), `samp11K` (11.025 kHz), `samp8K` (8.82 kHz), and `samp7K` (7.35 kHz).

Return Value: The current sampling rate.
See Also: audio_sample_rate(audio_sample_type rate)

7.9

void **mmxsnmp::audio_sample_rate** (audio_sample_type rate)

Sets the audio sampling rate

Sets the rate at which analog audio signals are processed by the audio codec. Legal sampling rates are `samp44K` (44.1 kHz), `samp29K` (29.4 kHz), `samp22K` (22.05 kHz), `samp17K` (17.64 kHz), `samp14K` (14.7 kHz), `samp11K` (11.025 kHz), `samp8K` (8.82 kHz), and `samp7K` (7.35 kHz).

Parameters: `rate` — The desired sampling rate.
See Also: audio_sample_rate()

7.10

int **mmxsnmp::audio_master_volume** ()

Reports the current master audio volume

Reports the current master output volume of the audio codec on both audio output channels A and B. The volume (*vol*) is specified as a percentage of the maximum volume, in the range $0 \leq vol \leq 100$; 0 means the audio is off, and 100 means the audio is being played at the maximum volume.

Return Value: The current master audio volume.
See Also: audio_master_volume(int vol)

7.11

void **mmxsnmp::audio_master_volume** (int vol)

Sets the master audio volume

Sets the master output volume of the audio codec on both audio output channels A and B. The volume (vol) must be specified as the percentage of the maximum volume, in the range $0 \leq vol \leq 100$; 0 turns audio off, and 100 plays the audio at the maximum volume. Values specified outside of this range will be silently rounded to the nearest legal value.

Parameters: `vol` — The desired master audio volume, $0 \leq vol \leq 100$.
See Also: audio_master_volume()

7.12

int **mmxsnmp::audio_feedback_volume** ()

Reports the current feedback volume

Reports the current feedback volume of the audio codec on both audio output channels A and B. The feedback volume is the percentage of the local audio input that is replayed on the local audio output. Some feedback is desirable when the user is wearing headphones. The volume (vol) is be specified as the percentage of the maximum volume, in the range $0 \leq vol \leq 100$; 0 means that feedback is off, and 100 means that all of the audio input is being replayed on the audio output.

Return Value: The current feedback volume.
See Also: audio_feedback_volume(int)

7.13

void **mmxsnmp::audio_feedback_volume** (int vol)

Sets the feedback volume

Sets the feedback volume of the audio codec on both audio output channels A and B. The feedback volume is the percentage of the local audio input that is replayed on the local audio output. Some feedback is desirable when the user is wearing headphones. The volume (*vol*) must be specified as the percentage of the maximum volume, in the range $0 \leq vol \leq 100$; 0 turns feedback off, and 100 replays all of the audio input back to the audio output. Values specified outside of this range will be silently rounded to the nearest legal value.

Parameters: `vol` — The desired feedback volume, $0 \leq vol \leq 100$.
See Also: audio_feedback_volume()

7.14

void **mmxsnmp::audio_mix** (int AatmAout, int AatmBout, int BatmAout, int BatmBout)

Sets the default audio mix table for future audio streams

Sets the default audio mix table to be used for future audio receive streams. Note that this function only changes the defaults and does not affect any streams that are currently active.

The mix table is specified as four volume parameters, each of which is in the range $0 \leq vol \leq 100$; 0 turns audio off, and 100 plays the audio at maximum volume. Values specified outside of this range will be silently rounded to the nearest legal value.

Parameters:
`AatmAout` — The percentage of audio received on channel A played on output A, $0 \leq AatmAout \leq 100$.
`AatmBout` — The percentage of audio received on channel A played on output B, $0 \leq AatmBout \leq 100$.
`BatmAout` — The percentage of audio received on channel B played on output A, $0 \leq BatmAout \leq 100$.
`BatmBout` — The percentage of audio received on channel B played on output B, $0 \leq BatmBout \leq 100$.

See Also: audio_mix(vc audio, int AatmAout, int AatmBout, int BatmAout, int BatmBout)

7.15

void **mmxsnmp::audio_mix** (vc audio, int AatmAout, int AatmBout, int BatmAout, int BatmBout)

Sets the current audio mix table on audio streams

Sets the audio mix table for one or all active audio streams. The mix table is specified as four volume parameters, each of which is in the range $0 \leq vol \leq 100$; 0 turns audio off, and 100 plays the audio at maximum volume. Values specified outside of this range will be silently rounded to the nearest legal value.

If there is an active audio receive stream with VPI `audio.vpi()` and VCI `audio.vci()`, that stream is given the specified mix table. If `audio` is the constant `mmxsnmp::idle_vc`, all active audio receive streams are given the specified mix table. If `audio` specifies an audio receive stream and no such stream is active, this function has no effect.

This function does not change the default mix table given to new audio receive connections.

Parameters:

`audio` — An instance of class `vc` containing the desired VPI and VCI of the audio receive stream to be modified, or the constant `mmxsnmp::idle_vc`.

`AatmAout` — The percentage of audio received on channel A played on output A, $0 \leq AatmAout \leq 100$.

`AatmBout` — The percentage of audio received on channel A played on output B, $0 \leq AatmBout \leq 100$.

`BatmAout` — The percentage of audio received on channel B played on output A, $0 \leq BatmAout \leq 100$.

`BatmBout` — The percentage of audio received on channel B played on output B, $0 \leq BatmBout \leq 100$.

See Also:

audio_mix(int AatmAout, int AatmBout, int BatmAout, int BatmBout)

7.16

mute_type **mmxsnmp::audio_mute_tx** ()

Reports the current muting of transmitted audio

Reports the current muting of the incoming signals on local audio inputs A and B.

When set to `MUTE_ON`, no signal is passed to the audio processor on the codec. The audio codec may continue to transmit (empty) audio cells depending on its current processing mode (i.e., `MODE0`).

When set to `MUTE_OFF`, the audio signal is encoded into ATM cells and transmitted to the ATM interface.

Return Value: The current TX audio mute state.
See Also: audio_mute_tx(mute_type m)

7.17

void **mmxsnmp::audio_mute_tx** (mute_type m)

Controls the muting of transmitted audio

Controls the muting of the incoming signals on local audio inputs A and B.

When set to `MUTE_ON`, no signal is passed to the audio processor on the codec. The audio codec may continue to transmit (empty) audio cells depending on its current processing mode (i.e., `MODE0`). If an ATM connection is established when the input is muted, it is unaffected.

When set to `MUTE_OFF`, the audio signal is encoded into ATM cells and transmitted to the ATM interface.

Parameters: `m` — The desired TX audio mute state: `MUTE_ON` or `MUTE_OFF`.
See Also: audio_mute_tx()

7.18

mute_type **mmxsnmp::audio_mute_rx** ()

Reports the current muting of received audio

Reports the current state of playback of audio to the analog outputs.

When set to `MUTE_ON`, no signal is provided on the analog outputs, but audio processing is active.

When set to `MUTE_OFF`, all received audio (both local and ATM) is mixed and provided on the analog outputs.

Return Value: The current RX audio mute state.
See Also: audio_mute_rx(mute_type m)

7.19

void **mmxsnmp::audio_mute_rx** (mute_type m)

Controls the muting of received audio

Controls the playback of audio to the analog outputs.

When set to `MUTE_ON`, no signal is provided on the analog outputs, but audio processing is active.

When set to `MUTE_OFF`, all received audio (both local and ATM) is mixed and provided on the analog outputs.

Parameters: `m` — The desired RX audio mute state: `MUTE_ON` or `MUTE_OFF`.
See Also: audio_mute_rx()

7.20

mute_type **mmxsnmp::audio_mute_self** ()

Reports the current state of "Self–Muting"

Reports the current state of "Self-Muting" performed by the MMX. If the audio codec is operating in `MODE1` and the self-mute flag is set to `MUTE_ON`, the user's audio stream is subtracted out of all the merged audio streams being received. If the flag is set to `MUTE_OFF`, this subtraction is not performed.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value: The current audio self-mute state.
See Also: audio_mute_self(mute_type m)

7.21

void **mmxsnmp::audio_mute_self** (mute_type m)

Controls "Self–Muting"

Sets the state of "Self-Muting" performed by the MMX. If the audio codec is operating in `MODE1` and the self-mute flag is set to `MUTE_ON`, the user's audio stream is subtracted out of all the merged audio streams being received. If the flag is set to `MUTE_OFF`, this subtraction is not performed.

When the audio codec is operating in `MODE0`, this function has no effect.

Parameters: `m` — The desired audio self-mute state: `MUTE_ON` or `MUTE_OFF`.
See Also: audio_mute_self()

7.22

int **mmxsnmp::audio_local_box_id** ()

Reports the local audio codec's Box ID

When the audio codec is operating in `MODE1`, reports the Box ID used to identify the audio stream generated by the local codec.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value: The audio codec Box ID.

7.23

int **mmxsnmp::audio_local_src_id** ()

Reports the local audio codec's Source ID

When the audio codec is operating in `MODE1`, reports the Source ID used to identify the audio stream generated by the local codec.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value: The current audio codec Source ID.
See Also: audio_local_src_id(int id)

7.24

void **mmxsnmp::audio_local_src_id** (int id)

Sets the local audio codec's Source ID

When the audio codec is operating in `MODE1`, sets the Source ID used to identify the audio stream generated by the local codec. The Source ID (id) must be in the range $0 \leq id \leq 65535$. Values specified outside of this range will be silently rounded to the nearest legal value.

When the audio codec is operating in `MODE0`, this function has no effect.

Parameters: `id` — The desired audio codec Source ID, $0 \leq id \leq 65535$.
See Also: audio_local_src_id()

7.25

int **mmxsnmp::audio_box_id_a** ()

Reports the Box ID of a received audio stream

When the audio codec is operating in `MODE1`, reports the Box ID of one of the two received audio streams currently being sent to the analog outputs.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value: The Box ID of received audio stream A.

7.26

int **mmxsnmp::audio_src_id_a** ()

Reports the Source ID of a received audio stream

When the audio codec is operating in `MODE1`, reports the Source ID of one of the two received audio streams currently being sent to the analog outputs.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value: The Source ID of received audio stream A.

7.27

int **mmxsnmp::audio_box_id_b** ()

Reports the Box ID of a received audio stream

When the audio codec is operating in `MODE1`, reports the Box ID of one of the two received audio streams currently being sent to the analog outputs.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value: The Box ID of received audio stream B.

7.28

int **mmxsnmp::audio_src_id_b** ()

Reports the Source ID of a received audio stream

When the audio codec is operating in `MODE1`, reports the Source ID of one of the two received audio streams currently being sent to the analog outputs.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value: The Source ID of received audio stream B.

7.29

int **mmxsnmp::audio_vox_thresh** ()

Reports the current VOX Threshold

When the audio codec is operating in `MODE1`, reports the current VOX threshold used to determine when to switch audio receive streams. The threshold ($thresh$) is in the range $0 \leq thresh \leq 65535$.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value: The current audio VOX threshold.
See Also: audio_vox_thresh(int thresh)

7.30

void **mmxsnmp::audio_vox_thresh** (int thresh)

Sets the VOX Threshold

When the audio codec is operating in `MODE1`, sets the VOX threshold used to determine when to switch audio receive streams. The threshold ($thresh$) must be in the range $0 \leq thresh \leq 65535$. Values specified outside of this range will be silently rounded to the nearest legal value.

When the audio codec is operating in `MODE0`, this function has no effect.

Parameters:	`thresh` — The desired audio VOX threshold, $0 \leq thresh \leq 65535$.
See Also:	audio_vox_thresh()
	audio_vox_hold(int t)

7.31

int **mmxsnmp::audio_vox_hold** ()

Reports the current VOX Hold Time

When the audio codec is operating in `MODE1`, reports the current VOX hold time, used to determine how long an audio stream must remain above the VOX threshold before the audio receive stream will switch. The VOX hold time (t) is the range $0 \leq t \leq 255$.

When the audio codec is operating in `MODE0`, this value has no meaning.

Return Value:	The current audio VOX hold time.
See Also:	audio_vox_hold(int t)

7.32

void **mmxsnmp::audio_vox_hold** (int t)

Sets the VOX Hold Time

When the audio codec is operating in `MODE1`, sets the VOX hold time, used to determine how long an audio stream must remain above the VOX threshold before the audio receive stream will switch. The VOX hold time (t) must be in the range $0 \leq t \leq 255$. Values specified outside of this range will be silently rounded to the nearest legal value.

When the audio codec is operating in `MODE0`, this function has no effect.

Parameters: `t` — The desired audio VOX hold time, $0 \leq t \leq 255$.

See Also: audio_vox_hold()
audio_vox_thresh(int thresh)

8 Video Functions

These functions control the operation of the video codec subsystem of the MMX.

8.1 void **mmxsnmp::video_start** (vc new_video)

Starts transmission or reception of a video stream

If `new_video.dir()` is `TRANSMIT`, the video codec will begin transmitting video on VPI `new_video.vpi()` and VCI `new_video.vci()`, from the default video input source, using the default Transmit Q Factor. If another video transmit stream is already active, that stream will automatically be closed.

If `new_video.dir()` is `RECEIVE`, the video codec will begin receiving video on VPI `new_video.vpi()` and VCI `new_video.vci()`, using the Default Receive Q Factor. If another video receive stream is already active, that stream will automatically be closed.

The Default Transmit and Receive Q Factors are both $Q = 30$ until some other value is specified using `video_q_factor(int Q, dir_type dir)`.

Parameters: `new_video` — An instance of class `vc` containing the desired VPI, VCI and direction of the new video stream. `new_video.dir()` must be `TRANSMIT` or `RECEIVE`.

See Also:
video_input(video_input_type v)
video_q_factor(int Q, dir_type dir)
video_stop(dir_type dir)
video_status(dir_type dir)

8.2

void **mmxsnmp::video_stop** (dir_type dir)

Stops video from being transmitted and/or received

If `dir` is `TRANSMIT` and a video transmit stream is active, the stream is closed. If `dir` is `RECEIVE` and a video receive stream is active, the stream is closed. If `dir` is `BIDIR`, both of the above operations are performed.

Parameters: `dir` — The direction of the video stream to be closed: `TRANSMIT`, `RECEIVE`, or `BIDIR`.
See Also: video_start(vc new_video)
video_status(dir_type dir)

8.3

vc **mmxsnmp::video_status** (dir_type dir)

Reports active video streams

Reports the VPI and VCI (through an instance of class `vc`) of active video streams.

If `dir` is `TRANSMIT` and there is an active video transmit stream, it is returned. If no video transmit stream is active, the constant `mmxsnmp::idle_vc` is returned.

If `dir` is `RECEIVE` and there is an active video receive stream, it is returned. If no video receive stream is active, the constant `mmxsnmp::idle_vc` is returned.

Return Value: The active video stream or the constant `mmxsnmp::idle_vc`.
Parameters: `dir` — `TRANSMIT` or `RECEIVE`.
See Also: video_start(vc new_video)
video_stop(dir_type dir)

8.4

int **mmxsnmp::video_q_factor** (dir_type dir)

Reports the currently selected Q Factor

If `dir` is `TRANSMIT` and there is an active video transmit stream, the Q Factor of that stream is reported. If no video transmit stream is active, the Default Transmit Q Factor (the Q Factor to be used for future video transmit streams) is reported.

If `dir` is `RECEIVE` and there is an active video receive stream, the Q Factor of that stream is reported. If no video receive stream is active, the Default Receive Q Factor (the Q Factor to be used for future video receive streams) is reported.

The Q Factor is in the range $10 \leq Q \leq 1000$.

Return Value: The requested Q Factor.
Parameters: `dir` — `TRANSMIT` or `RECEIVE`.
See Also: video_q_factor(int Q, dir_type dir)

8.5

void **mmxsnmp::video_q_factor** (int Q, dir_type dir)

Sets the active and default Q Factor

The Q Factor controls the JPEG compression and decompression algorithm applied to each frame of transmitted and received video. Higher Q Factors result in more compression, lower required bandwidth for the video stream, and greater degradation of the image. Lower Q Factors result in less compression, higher bandwidth required for the video stream, and less degradation of the image. The actual compression achieved and bandwidth required are dependent on other factors such as the complexity of the image and the amount of motion within the field of view.

For a video stream to be properly received, the video transmit and video receive streams must be set to the same Q Factor. It is the responsibility of the application to ensure that the synchronization of Q Factors is maintained. "Asymmetrical" video streams are allowed; for example, it is legal for MMX M_1 to be transmitting video to MMX M_2 at $Q = 30$ and receiving at $Q = 40$ while M_2 is transmitting video to M_1 at $Q = 40$ and receiving at $Q = 30$.

If `dir` is `TRANSMIT`, the Default Transmit Q Factor (the Q Factor to be used for future video transmit streams) is set to `Q`. If there is an active video transmit stream, its Q Factor is also changed to `Q`.

If `dir` is `RECEIVE`, the Default Receive Q Factor (the Q Factor to be used for future video receive streams) is set to `Q`. If there is an active video receive stream, its Q Factor is also changed to `Q`.

If `dir` is `BIDIR`, the `TRANSMIT` and `RECEIVE` operations are both performed.

Currently, the library provides no method for setting the Default Q Factor without also changing the Q Factor of active video streams, or vice-versa.

Legal values for Q are in the range $10 \leq Q \leq 1000$; values specified outside of this range will be silently rounded to the nearest legal value. A typical value is $Q = 30$. The Default Transmit and Receive Q Factors are both 30 until some other value is specified.

Parameters: `dir` — `TRANSMIT`, `RECEIVE`, or `BIDIR`.
`Q` — The desired Q Factor, $10 \leq Q \leq 1000$.

See Also: video_q_factor(dir_type dir)

8.6

video_input_type **mmxsnmp::video_input** ()

Reports the current video input source

If there is an video transmit stream, the video input of that stream is reported. If no video transmit stream is active, the default video input (the input to be used for future video transmit streams) is reported.

Legal values are `CIN1` (the first composite video source), `CIN2` (the second composite video source), and `SIN1` (the first S-Video input source).

Return Value:	The active or default video source.
See Also:	video_input(video_input_type v)

8.7

void **mmxsnmp::video_input** (video_input_type v)

Sets the active and default video input source

The default video input (the input to be used for future video transmit streams) is changed to `v`. If there is an active video transmit stream, its video input source is also changed to `v`.

Currently, the library provides no method for setting the default video input source without also changing the video input source of active streams, or vice-versa.

Legal values are `CIN1` (the first composite video source), `CIN2` (the second composite video source), and `SIN1` (the first S-Video input source).

Parameters:	`v` — The desired video input source: `CIN1`, `CIN2`, or `SIN1`.
See Also:	video_input()

8.8

mute_type **mmxsnmp::video_mute_tx** ()

Reports the current muting of transmitted video

If set to `MUTE_ON` and a video transmit stream has been established, a blank screen is transmitted. If set to `MUTE_OFF`, the incoming video signal on the currently selected video input is transmitted.

Return Value:	The current video TX mute state.
See Also:	video_mute_tx(mute_type m)

8.9

void **mmxsnmp::video_mute_tx** (mute_type m)

Controls the muting of transmitted video

If set to `MUTE_ON` and a video transmit stream has been established, a blank screen is transmitted. If set to `MUTE_OFF`, the incoming video signal on the currently selected video input is transmitted.

Parameters:	`m` — The desired video TX mute state: `MUTE_ON` or `MUTE_OFF`.
See Also:	video_mute_tx()

8.10

pll_type **mmxsnmp::video_pll_mode** ()

Reports the currently selected Video PLL Mode

The PLL Mode describes the type of video source connected to the system. This setting is required for proper synchronization between the video source and the video codec. Legal values are `CAMERA` and `VTR`.

Return Value:	The currently selected Video PLL Mode.
See Also:	video_pll_mode(pll_type pll)

8.11

void **mmxsnmp::video_pll_mode** (pll_type pll)

Selects a Video PLL Mode

The PLL Mode describes the type of video source connected to the system. This setting is required for proper synchronization between the video source and the video codec. Legal values are `CAMERA` and `VTR`.

Parameters: `pll` — The desired Video PLL Mode: `CAMERA` or `VTR`.
See Also: video_pll_mode()

8.12

video_format_type **mmxsnmp::video_format** ()

Reports the system's video format

Legal values are `NTSC`, `PAL`, and `OTHER`.

Return Value: The system's video format.